GENE EXPRESSION PROGRAMMING FOR LOGIC CIRCUIT DESIGN

by

STEVEN MANDLA MASIMULA

submitted in accordance with the requirements for
the degree of

MASTER OF SCIENCE

In the subject

APPLIED MATHEMATICS

at the

University of South Africa

Supervisor: Professor Yorick Hardy

3 October 2017

Declaration

Student number: **4236-479-5**

I declare that **GENE EXPRESSION PROGRAMMING FOR LOGIC CIRCUIT DESIGN** is my own work and that all the sources that I have used or quoted have been indicated and acknowledged by means of complete references.

SIGNATURE
(Mr S M Masimula)

DATE

Acknowledgements

I wish to extend my sincere appreciation and gratitude to the following people for their role and contribution towards my studies:

- Professor Yorick Hardy my supervisor for his tremendous patience, guidance and supervision.
- My family, friends and colleagues for their support.

Abstract

Finding an optimal solution for the logic circuit design problem is challenging and time-consuming especially for complex logic circuits. As the number of logic gates increases the task of designing optimal logic circuits extends beyond human capability. A number of evolutionary algorithms have been invented to tackle a range of optimisation problems, including logic circuit design. This dissertation explores two of these evolutionary algorithms i.e. Gene Expression Programming (GEP) and Multi Expression Programming (MEP) with the aim of integrating their strengths into a new Genetic Programming (GP) algorithm. GEP was invented by Candida Ferreira in 1999 and published in 2001 [8]. The GEP algorithm inherits the advantages of the Genetic Algorithm (GA) and GP, and it uses a simple encoding method to solve complex problems [6, 32]. While GEP emerged as powerful due to its simplicity in implementation and flexibility in genetic operations, it is not without weaknesses. Some of these inherent weaknesses are discussed in [1, 6, 21]. Like GEP, MEP is a GP-variant that uses linear chromosomes of fixed length [23]. A unique feature of MEP is its ability to store multiple solutions of a problem in a single chromosome. MEP also has an ability to implement code-reuse which is achieved through its representation which allow multiple references to a single sub-structure.

This dissertation proposes a new GP algorithm, Improved Gene Expression Programming (IGEP) which improves the performance of the traditional GEP by combining the code-reuse capability and simplicity of gene encoding method from MEP and GEP, respectively. The results obtained using the IGEP and the traditional GEP show that the two algorithms are comparable in terms of the success rate when applied on simple problems such as basic logic functions. However, for complex problems such as one-bit Full Adder (FA) and AND-OR Arithmetic Logic Unit (ALU) the IGEP performs better than the traditional GEP due to the code-reuse in IGEP.

Key terms:

Logic circuit design; Genetic algorithms; Genetic programming; Gene expression programming; Multi expression programming; Improved gene expression programming; Improved multi expression programming; Cartesian genetic programming; Automatically defined function; Multi-expression based Gene expression programming.

List of Figures

List of Tables

List of Acronyms

GEP Gene Expression Programming

MEP Multi Expression Programming

IGEP Improved Gene Expression Programming

GP Genetic Programming

ET expression tree

IS Insertion Sequence

RIS Root Insertion Sequence

CGP Cartesian Genetic Programming

ADF Automatically Defined Function

IMEP Improved Multi Expression Programming

ALU Arithmetic Logic Unit

HA Half Adder

FA Full Adder

GA Genetic Algorithm

MGEP Multi-expression based Gene Expression Programming

Contents

Chapter 1

Introduction and background

1.1 Introduction

Savage [27, p. 35] defines a logic circuit, the basic building block of real-world computers, as a circuit in which the operations are boolean. The design of a logic circuit is the process of determining, from input/output behaviour specification, a structure (a combination of logic gates) that is functional such that, for given inputs, the structure implements a given truth table. Above all, this design must be as optimal as possible in terms of specified constraints (e.g. the number of gates) [5]. The design of functional logic circuits (especially complex ones) can be cumbersome for human designers, let alone designing optimum logic circuits. On this note, various genetic algorithms for automation of the logic circuit design process have been developed. However, the focus is and has always been on the performance of these algorithms. This research explores the application of one type of genetic algorithms, GEP, in the area of logic circuit design. Most importantly, in this study the efficiency of the GEP algorithm is enhanced by borrowing strength from another genetic algorithm called MEP [23, 31]. The enhanced GEP algorithm is referred to as IGEP algorithm.

Section 2.1 of this dissertation outlines the GEP while the MEP and IGEP algorithms are discussed in Sections 2.2 and 2.3, respectively. Literature review is conducted in Chapter 3 and the results of nine case studies are presented in Chapter 4. Chapter 5 concludes the dissertation.

1.2 Statement of the problem

More formally, the focus of this research is to design a functional optimum logic circuit that performs a desired function (specified by a truth table), given a certain specified set of logic gates using IGEP, the modified GEP algorithm. The complexity of a logic circuit is a function of the number of gates in the circuit. The complexity of a logic gate is generally the number of inputs to it.

1.3 Background of study

As the scale of logic circuit design increases (i.e. increasing number of logic gates), the problem of logic circuit design, let alone determining the optimality of the resulting circuit, extends beyond human capability. It is expected that the results of this research will assist in terms of simplifying the design of logic circuits and thus improve on the time it takes to produce optimal logic circuits. This research focuses on the implementation

using an improved GEP, IGEP, which integrates reuse of common sub-structures (genes) into the traditional GEP. IGEP borrows strength from MEP's sub-expression reuse feature. With this feature, the MEP is able to repeatedly use the same sub-expression (common sub-structure) in an expression without repeating it. This feature improves the efficiency of the MEP algorithm as it takes shorter time to evaluate chromosomes since repeated sub-structures are evaluated only once and reused in the subsequent stages of evaluation.

1.4 Objectives of study

The main objectives of this research are:

- To explore the GEP and MEP techniques.
- To demonstrate how GEP can be applied in the area of logic circuit design.
- To improve the efficiency of GEP by reusing sub-structures (genes).

Chapter 2

Gene expression programming and multi expression programming

2.1 Gene expression programming

GEP was invented in 1999 and appeared in a publication for the first time in 2001 by Candida Ferreira [8]. GEP uses the same kind of diagram representation of GP, but the entities produced by GEP are the expression of genome (ETs). GEP led to the invention of chromosomes capable of representing any ET. Like genetic algorithms and GP, GEP is a genetic algorithm as it uses populations of individuals, selects them according to fitness, and introduces genetic variation using one or more genetic operators [8]. The main building blocks of GEP are chromosomes and the ETs. For that purpose a language called Karva [8] was created to read and express the information of GEP chromosome. Furthermore, the structure of chromosomes was designed to allow the creation of multiple genes, each encoding a sub-ET. The genes are structurally composed of a head and tail and it is this structural and functional organisation that always guarantees the production of valid chromosome, no matter how much or how profoundly the chromosomes are modified.

The ETs express the genetic information encoded in the chromosomes. Translating chromosomes to ETs requires some kind of code and rules. The genetic code is very simple: a one-to-one relationship between symbols and functions or the terminals they represent. The rules are also simple: they determine the spatial organisation of the functions and terminals in the ETs and the type of interaction between the sub-ETs in multi-genic systems [8].

In the context of the Karva language, given the chromosome (genotype), the phenotype can easily be represented by an ET for example as shown in Figure 2.1. For this example, a set of functions F = {&, | } and the set of terminals T = {a, b, c} are used, where symbols | and & represent logical functions OR and AND, respectively. The example shows one-genic chromosome in the form of a logic expression, GEP prefix representation and an ET. The bold part is the *head* and remaining part is the *tail*.

Logic expression: (b&c)|(a&(b&c))

Prefix representation of GEP chromosome: |**&bc&a&bc**acbcccabbc

Expression tree:

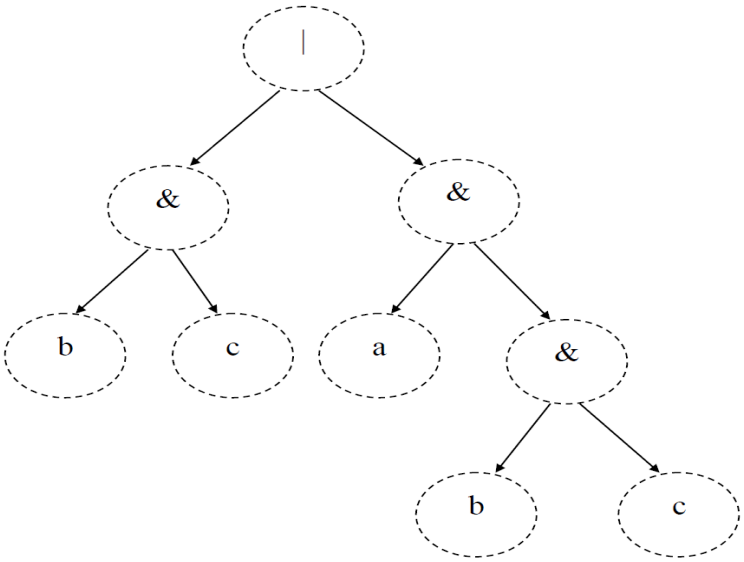

Figure 2.1: An example of GEP ET representing a logic expression: |**&bc&a&bc**acbcccabbc.

In this example, the bold part of the GEP chromosome is the head and it also coincides with what is called K-expression in Karva language. K-expression is the valid segment of the chromosome obtained by reading the ET from left to right and from top to bottom; and it can be represented as an ET shown in Figure 2.1.

2.1.1 Gene representation

GEP genes are composed of a head and a tail. The head contains symbols that represent both functions and terminals, whereas the tail contains only terminals. For each problem, the length of the head h is chosen, whereas the length of the tail t is a function of h and the number of arguments of the function with most arguments n (arity), and is given by [8, 9]:

$$t = h(n-1) + 1 \tag{2.1}$$

GEP chromosomes are usually composed of more than one gene of equal length. For each problem or run, the number of genes, as well as the length of the head, are *a priori* chosen. Each gene codes for sub-ET and the sub-ETs interact with one another forming a more complex multi-subunit ET [8, 9].

Figure 2.1 shows a one-genic chromosome of length 19. The head has length h=9 and the arity is n=2 which gives a tail of length t=10.

2.1.2 Genetic operators

Genetic operators are the core of all genetic algorithms and two of them are common to all evolutionary systems i.e. selection and replication. The following operators (discussed in detail in [8] and summarised in [30, p. 455])

are used in GEP to evolve or introduce genetic variation into the population:

a. selection, replication and elitism

b. mutation

c. transposition of IS elements

d. root transposition

e. gene transposition

f. recombination - this is a crossover operation. It can take any of the three forms:

   - one-point recombination
   - two-point recombination
   - gene recombination.

In [1], the GEP algorithm is summarised as shown in Figure 2.2, below:

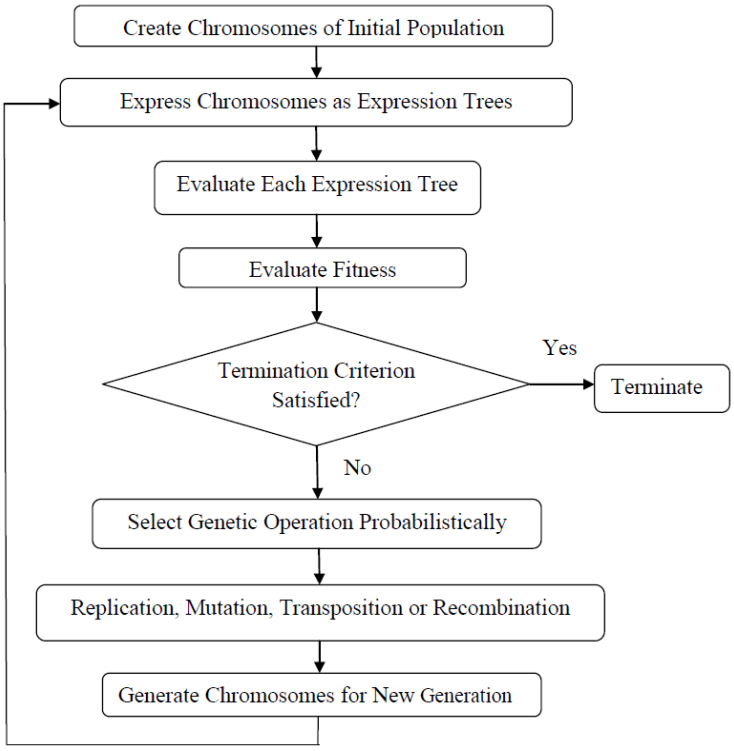

Figure 2.2: Flowchart of GEP algorithm

Each of these operators will be discussed in detail in the next sections.

2.1.2.1 Selection, replication and elitism

Selection is the process of determining the number of times a particular individual is chosen for reproduction and, thus, the number of offspring that an individual will produce. The principle behind genetic algorithms is essentially Darwinian natural selection. There is a number of selection methods that one can choose from e.g. ranking selection, tournament selection and roulette wheel (fitness-proportional) selection. However, the roulette wheel selection is a popular selection method. In *rank selection*, selection is based on the rank (not

the numerical value) of the fitness values of the individuals in the population while in *tournament selection*, a specified group of individuals (typically two) are chosen at random from the population and the one with the better fitness (i.e., the lower standardised fitness) is then selected [15, p. 100]. The fitness value for each individual is calculated based on the fitness function. The detailed discussion on the fitness function can be found in Section 2.4. The *roulette wheel selection* probabilistically selects individuals based on their fitness values f_i such that $f_i > 0$. The probability that an individual (chromosome) i is selected, p_i, is computed as:

$$p_i = \frac{f_i}{\sum_{k=1}^{n} f_k}. \tag{2.2}$$

Since now $\sum_{i=1}^{n} p_i = 1$, this method allows the fitness values to be used as probabilities. Individuals are then mapped one-to-one into contiguous intervals proportionally to their fitness i.e. the size of each individual interval corresponds to the fitness value of the associated individual. The circumference of the roulette wheel is the sum of all fitness values of the individuals. The fittest individual occupies the largest interval or segment, whereas the least fit have correspondingly smaller intervals within the roulette wheel, see Figure 2.3.

Using the roulette wheel selection, parents are selected according to their fitness. The greater the fitness of a chromosome, the greater the chance of that chromosome being selected. To select two individuals for reproduction, two random numbers, r_1 and r_2 are generated such that $0 \leq r_1 \leq 1$ and $0 \leq r_2 \leq 1$. Two chromosomes g_i and g_j (where i and j are positions in the population) are chosen by means of the following criteria:

$$\sum_{u=1}^{i} p_u \geq r_1 \geq \sum_{u=1}^{i-1} p_u, \tag{2.3}$$

$$\sum_{v=1}^{j} p_v \geq r_2 \geq \sum_{v=1}^{j-1} p_v. \tag{2.4}$$

That is, a random number, r, is generated in the interval [0, 1] and the first population member whose probability of selection added to the preceding population members (cumulative probability of selection) is greater than or equal to r is selected. This process is repeated until the desired number of individuals has been selected.

Table 2.1 below shows an example of a roulette wheel. Consider a population of seven individuals with associated fitness and probability of selection proportional to fitness.

Table 2.1: An example of roulette wheel

Population (*i*)	Fitness (f_i)	p_i	Cumulative p_i
Individual 1	4	0.098	0.098
Individual 2	8	0.195	0.293
Individual 3	5	0.122	0.415
Individual 4	6	0.146	0.561
Individual 5	7	0.171	0.732
Individual 6	8	0.195	0.927
Individual 7	3	0.073	1.000
SUM	**41**	**1.000**	

Graphically, the roulette wheel can be represented as a wheel in which segments are of possibly different sizes, based on each individual's relative fitness, see Figure 2.3 below.

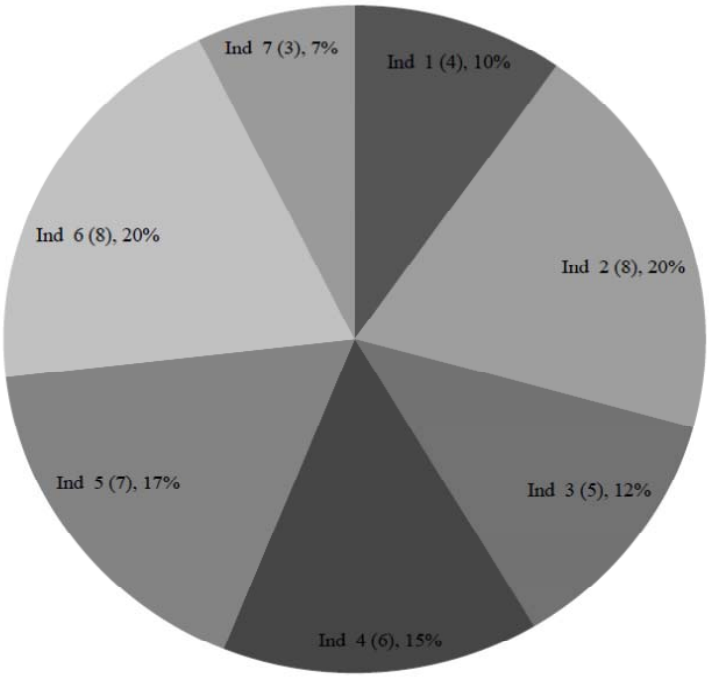

Figure 2.3: An example of the roulette wheel representation

To select two individuals for reproduction from the population shown in Table 2.1, suppose two random numbers, $r_1 = 0.814$ and $r_2 = 0.635$ are generated. Using r_1 the first parent chromosome to be selected for reproduction is 'Individual 6' as $\sum_{v=1}^{6} p_v = 0.927 \geq r_1 = 0.814$. The second parent chromosome to be selected is 'Individual 5' as $\sum_{v=1}^{5} p_u = 0.732 \geq r_2 = 0.635$. It should be noted that in this study the lower the fitness value the fitter the individual, hence the roulette wheel is applied on the inverse of the fitness value.

It is worth noting that probabilistic operations enter the genetic algorithms in three different phases. First, the initial population must be selected. This choice can be made randomly (or if some prior knowledge of good starting points exist, these can be chosen). Next, members of the population have to be selected for reproduction. One way to do this is to select individuals probabilistically based on their fitness. The third way probabilities enter into consideration is in the selection of the genetic operation to be used.

As described in [26, 18], *elitism* is essentially a mechanism that protects the best chromosomes in subsequent generations. In classical genetic algorithms the best individuals are not always transfered to the next generation. It does not always happen that the next generation contains the fittest chromosomes from the current population. Elitism is applied in order to protect populations against the loss of fittest individuals as a result of genetic operators. The fittest individuals are always carried forward to the next generation unaltered and thus the minimum fitness of the population can never reduce from one generation to the next. Above all, elitism usually brings about a more rapid convergence of the population.

2.1.2.2 Mutation

This operation involves changing symbols in the chromosome but this is done such that the structure of the chromosome remains intact i.e. symbols in the tail of a gene may not operate on any arguments. For example, consider the following one-genic chromosome:

	0	**1**	**2**	**3**	**4**	**5**	**6**	7	8	9	10	11	12	13	14
Parent 1:	**a**	**&**	**b**	**c**	**a**	**&**	**c**	b	a	a	a	c	b	b	b

Suppose a mutation changed the **'&'** in position 1 to **'|'** and the **'c'** in position 6 to **'&'**, obtaining:

	0	**1**	**2**	**3**	**4**	**5**	**6**	7	8	9	10	11	12	13	14
Child 1:	**a**	**|**	**b**	**c**	**a**	**&**	**&**	b	a	a	a	c	b	b	b

It should be noted that if a function is mutated into a terminal or vice versa, or a function of one argument is mutated into a function of two arguments or vice versa, the ET is modified drastically [8]. Sometimes, mutation takes place in the non-coding region of the chromosome, leading to what is called neutral mutation.

2.1.2.3 Transposition of IS elements

A portion of a chromosome is chosen to be inserted in the head of a gene. The tail of the gene remains unaffected. Thus symbols are removed from the end of the head to make room for the inserted string [30]. The transposition operator randomly chooses the chromosome, the start of the IS element, the site, and the length of the transposon. A transposition rate of 0.1 is typically used. Consider the following one-genic parent chromosome:

	0	**1**	**2**	**3**	**4**	**5**	**6**	7	8	9	10	11	12	13	14
Parent 1:	**a**	**|**	**b**	**c**	**a**	**&**	**&**	b	a	a	a	c	b	b	b

Suppose that a sequence **'&baa'** in position 6 to 9 is chosen to be an IS element and inserted in the head starting from position 1, obtaining:

	0	**1**	**2**	**3**	**4**	**5**	**6**	7	8	9	10	11	12	13	14
Child 1:	**a**	**&**	b	a	a	**|**	**b**	b	a	a	a	c	b	b	b

2.1.2.4 Root transposition

All Root Insertion Sequence (RIS) elements start with functions, and thus are chosen among the sequences of the heads. For that, a point is randomly chosen in the head and the gene is scanned downstream until a function is found. This function becomes the start position of the RIS element. If no functions are found, the operator does nothing. Typically, a root transposition rate of 0.1 is used. As an example, consider the following one-genic parent chromosome:

	0	**1**	**2**	**3**	**4**	**5**	**6**	7	8	9	10	11	12	13	14
Parent 1:	|	**&**	**&**	|	**a**	|	**b**	c	a	b	a	c	b	b	b

Suppose the sequence **'|a'** in position 3 to 4 in the parent chromosome above is copied into the root of the chromosome, resulting in:

	0	**1**	**2**	**3**	**4**	**5**	**6**	7	8	9	10	11	12	13	14
Child 1:	|	**a**	|	**&**	**&**	|	**a**	c	a	b	a	c	b	b	b

2.1.2.5 Gene transposition

In gene transposition, an entire gene functions as a transposon and transposes itself to the beginning of the chromosome. In contrast with other forms of transposition, in gene transposition the transposon (the gene) is deleted in the place of the origin [1]. One gene (except the first) in a chromosome is randomly chosen to be the first gene. All other genes in the chromosome are shifted downwards in the chromosome to make place for the first gene. Consider the following two-genic chromosome:

	Gene 1									**Gene 2**								
	0	**1**	**2**	**3**	4	5	6	7	8	**0**	**1**	**2**	**3**	4	5	6	7	8
Parent 1:	|	**&**	**a**	**b**	a	a	b	b	a	**&**	**&**	|	**a**	a	b	b	b	b

In this example, gene 2 is selected for gene transposition by default since there are only two genes in the chromosome, giving:

	Gene 1									**Gene 2**								
	0	**1**	**2**	**3**	4	5	6	7	8	**0**	**1**	**2**	**3**	4	5	6	7	8
Child 1:	**&**	**&**	|	**a**	a	b	b	b	b	|	**&**	**a**	**b**	a	a	b	b	a

In case of a two-genic chromosome the operator essentially swaps the positions of the genes.

2.1.2.6 Recombination

In GEP, there are three kinds of recombination: one-point, two-point and gene recombination. In all cases, two parent chromosomes are randomly chosen and paired to exchange some genetic material between them.

a. One-point recombination

   In one-point recombination the chromosomes are paired and split in the same point and corresponding sections are swapped. Consider the following two one-genic parent chromosomes for one-point recombination:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Parent 1:	|	a	|	&	&	|	a	c	a	b	a	c	b	b	b
Parent 2:	|	**&**	**b**	**a**	**a**	|	**b**	**c**	**a**	**a**	**a**	**c**	**b**	**b**	**b**

Suppose that position 3 is chosen as recombination point, resulting in the following children:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Child 1:	|	a	|	**a**	**a**	|	**b**	**c**	**a**	**a**	**a**	**c**	**b**	**b**	**b**
Child 2:	|	**&**	**b**	&	&	|	a	c	a	b	a	c	b	b	b

b. Two-point recombination

In this operator, two parent chromosomes are split into three and the middle portion is swapped forming new children. The recombination points are chosen randomly, for example:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Parent 1:	|	a	|	&	&	|	a	c	a	b	a	c	b	b	b
Parent 2:	**&**	**&**	**b**	**a**	**a**	**&**	**b**	**c**	**a**	**a**	**a**	**c**	**b**	**b**	**b**

Suppose positions 3 and 10 are selected randomly such that the genetic material between positions 3 and 10 is swapped between the parent chromosomes to produce two children:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Child 1:	|	a	|	**a**	**a**	**&**	**b**	**c**	**a**	**a**	**a**	c	b	b	b
Child 2:	**&**	**&**	**b**	&	&	|	a	c	a	b	a	**c**	**b**	**b**	**b**

c. Gene recombination

In gene recombination, genes are exchanged between two parent chromosomes, forming two children containing genes from both parents. The exchanged genes are randomly chosen and occupy the same position in the parent chromosomes. Below is an example of two two-genic parent chromosomes:

	Gene 1									**Gene 2**								
	0	1	2	3	4	5	6	7	8	0	1	2	3	4	5	6	7	8
Parent 1:	&	&	|	c	a	b	b	a	b	|	&	a	b	c	a	b	c	c
Parent 2:	**&**	**&**	**b**	**&**	**a**	**c**	**c**	**b**	**c**	**&**	|	**a**	**b**	**a**	**a**	**c**	**b**	**a**

Suppose the second gene in parent chromosome 1 is chosen at random and swapped with the second gene in parent chromosome to obtain:

	Gene 1									**Gene 2**								
	0	1	2	3	4	5	6	7	8	0	1	2	3	4	5	6	7	8
Child 1:	&	&	|	c	a	b	b	a	b	**&**	|	**a**	**b**	**a**	**a**	**c**	**b**	**a**
Child 2:	**&**	**&**	**b**	**&**	**a**	**c**	**c**	**b**	**c**	|	&	a	b	c	a	b	c	c

The children contain entire genes from both parents. In this kind of recombination, similar genes can be exchanged but, most of the times, the exchanged genes are very different and new material is introduced in the population. However, with this operator, no new genes are created [8].

2.2 Multi expression programming

Like GEP, MEP is a GP variant that uses linear chromosomes of fixed length [23]. A unique feature of MEP is its ability to store multiple solutions of a problem in a single chromosome. Note that this feature does not increase the complexity of the MEP decoding process when compared to other techniques storing a single solution in a chromosome. Studies show that MEP performs better than other competitor techniques such as GEP and CGP [19] for some well-known problems such as symbolic regression [23]. As already indicated, GEP has been implemented successfully in the area of logic circuit design. However, the performance and efficiency of these algorithms has always been an issue despite the proposed improvements by various researchers. On this note, this research integrates the features of the MEP into GEP in order to improve the performance and efficiency of GEP, notably the ability of MEP to represent a common sub-structure in a chromosome without repeating it i.e. code-reuse. The code-reuse ability in MEP is achieved through its representation which allow multiple references to a single sub-structure.

2.2.1 Gene representation

The MEP representation ensures that no cycle arises while the chromosome is decoded. Figure 2.4 shows an example of MEP representation. According to this representation scheme the first symbol of the chromosome must be a terminal symbol. Each gene encodes a terminal or function symbol. A gene encoding a function includes pointers towards the function arguments which always point to expressions in earlier positions in the chromosome. In this way only syntactically correct MEP individuals are obtained since the translation of a MEP chromosome is done by reading the chromosome top-down with later expressions referencing earlier expressions via pointers [23]. The prefix MEP representation of the chromosome in Figure 2.1 is given below in Figure 2.4:

0: b
1: c
2: & 0, 1
3: a
4: & 2, 3
5: | 2, 4

Figure 2.4: An example of prefix MEP representation of the chromosome: |**&bc&a&bc**acbcccabbc.

Note that the sub-structure **'&bc'** is repeated in Figure 2.1. As can be seen, in the MEP representation this common sub-structure is not repeated. It is exactly this ability of MEP to repeatedly use the same sub-expression (common sub-structure) in an expression without repeating it that this research aims to exploit to the benefit of GEP as it takes shorter time to evaluate chromosomes; since repeated sub-structures are evaluated only once and the results are reused in the subsequent stages of evaluation.

MEP can be expressed by a directed graph of indexed nodes. The graph has a set of n_i inputs that are indexed as nodes 0 to $n_i - 1$, a set of n_n nodes and a set of n_o outputs. Each n_n node has a number of inputs and a function which computes an output based on the inputs as shown in Figure 2.5 [10]. The genotype is a list of integers that determine the connectivity and functionality of the nodes. As shown in [10], these can be mutated and crossed over to create new directed graphs.

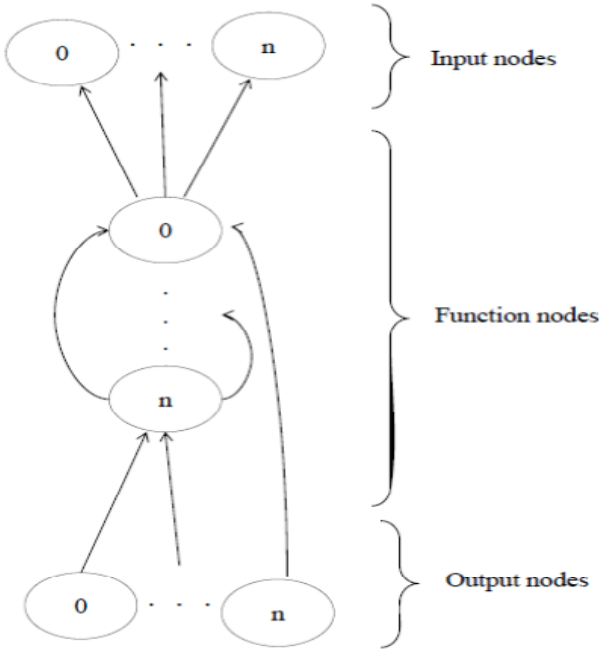

Figure 2.5: Graphical representation of MEP

As an example, the logic expression shown in Figure 2.4 translates to the directed graph shown in Figure 2.6.

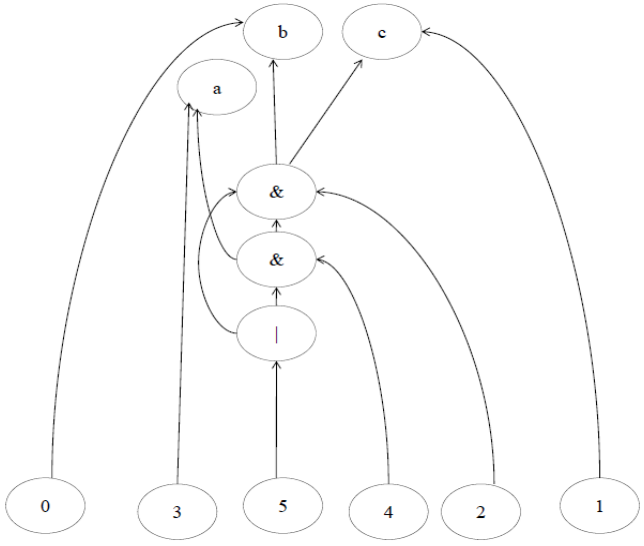

Figure 2.6: Graphical representation of the MEP logic expression shown in Figure 2.4

According to [23], using the above representation, a terminal symbol specifies a simple expression and a function symbol specifies a complex expression obtained by connecting the operands specified by the argument positions with the current symbol. The chromosome fitness is defined as the number of non-repeated evaluations in an expression e.g. there are six function evaluations in the prefix MEP representation of the chromosome shown above.

Notable differences between GEP and MEP [21, p. 12] are:

- The pointers toward function arguments are encoded explicitly in MEP chromosomes whereas in GEP these pointers are computed when chromosomes are parsed. MEP also needs to evolve the pointers toward function arguments. This leads to a more compact representation of GEP when compared to MEP.
- MEP representation is suitable for code-reuse, while the GEP is not.
- The non-coding regions of GEP are always at the end of the chromosome, whereas in MEP these regions can appear anywhere in the chromosome.

2.2.2 Genetic operators

Unlike GEP, two genetic operators are used in MEP: crossover and mutation. These operators preserve the chromosome structure. All offsprings obtained by crossover and mutation are always syntactically correct [20, p. 4]. Thus, no extra processing for repairing newly obtained individuals is needed [11, p. 105]. In [10], the author notes that applying genetic operators (i.e. mutation and crossover) to a graph structure is not simple since the integrity of the chromosome must not be compromised. In contrast, the mutation operator in GEP is simple as there is only one restriction i.e. symbols in the tail of a gene may not operate on any arguments.

2.2.2.1 Crossover

Crossover points are picked at random i.e. any point along the parent chromosome can be chosen and everything before this point goes to one child while everything after goes to the other child [10]. The crossover process involves the random selection of two parent chromosomes for recombination. For instance, within the uniform recombination the offspring genes are taken randomly from one parent or another. As an example, consider two parent chromosomes where positions 1, 2 and 5 are crossed:

Parent 1	**Parent 2**	**Child 1**	**Child 2**
0: b	0: a	**0: b**	0: a
1: c	1: b	1: b	**1: c**
2: & 0, 1	2: | 0, 1	2: | 0, 1	**2: & 0, 1**
3: a	3: & 0, 1	**3: a**	3: & 0, 1
4: & 3, 2	4: | 2, 3	**4: & 3, 2**	4: | 2, 3
5: | 2, 4	5: & 4, 0	5: & 4, 0	**5: | 2, 4**

2.2.2.2 Mutation

Determining which point to mutate is done by randomly choosing a point along the chromosome. Each symbol (terminal, function, or pointer) in the chromosome may be the target of the mutation operator [21]. If a pointer was chosen, it can only be mutated to a random value between 1 and $n-1$, where n is the current node while

functions can be mutated to any function (and arguments) in the function set and a terminal or an expression can mutate to some value in the terminal set or any expression with valid pointers i.e. the pointers may only point to nodes that come before the node that is being mutated. Most importantly, to preserve the consistency of the chromosome its first gene must encode a terminal symbol [10, 21]. Consider the following example, whereby the bold symbols in a parent chromosome are selected for mutation:

Parent	**Child**
0: a	0: a
1: b	**1:& 0, 0**
2: | 0, 1	2:& 0, 1
3: c	3: c
4: & 3, 2	**4: | 3, 2**
5: & 4, 0	5: & 4, 0

As an example, from the above figure, pointer 2 can only be mutated to either pointer 0 or 1. Function (&) in pointer 4 can be mutated to any function in the function set and a terminal (e.g. b in pointer 1) or an expression (e.g. & 4, 0 in pointer 5) can mutate to any value in the terminal set or any expression with valid pointers i.e. the pointers may only point to nodes that come before the node that is being mutated

2.3 Improved gene expression programming

GEP tends to provide simplicity in implementation and a flexibility in genetic operations compared to the other methods described in this dissertation. We improve on these strengths while largely preserving the simplicity of the implementation and flexibility of genetic operations.

In terms of the basic structure and application of genetic operators, the IGEP is the same as the traditional GEP; hence the details that were discussed in Section 2.1 are not repeated here. This section highlights only the improvements in IGEP. The only difference between GEP and IGEP is the MEP-like sub-structure (gene) reuse ability of IGEP which was implemented to improve efficiency. As with Automatically Defined Functions (ADFs), from an implementation point of view, the introduction of reusable genes adds an extra step to the preparatory steps from the GEP application. The extra step defines the architecture in terms of number of genes to be reused [24, p. 24]. However, gene-reuse greatly reduces the effort required in solving large complex problems while increasing the robustness of IGEP. The IGEP algorithm can be summarised as follows:

Let:
T = terminal set
n = number of genes
N = population size

T denotes the set of terminals, however the set of terminals differs according to the gene position in the chromosome. A later gene may refer to an earlier gene i via the terminal symbol i. Gene 1 begins with a base terminal set T_0 which does not reference any other gene.

Step 1: **Population initialisation**

for $j = 1$ **to** N **do**
  $T \leftarrow T_0$ (initialise the terminal set);
  for $i = 1$ **to** n **do**
    randomly generate gene i according to the GEP rules;
    $T \leftarrow T \cup \{i\}$
  end
end

Step 2: **GEP algorithm**

2.1: Apply selection based on roulette wheel selection and elitism algorithms such that 10% of the current population is transfered to the next generation (including the top 5% best chromosomes).

2.2: Apply genetic operators. The parent chromosomes are selected for reproduction using the roulette wheel selection.

2.3: Evaluate current population.

2.4: While the maximum number of iterations is not reached, repeat steps 2.1 to 2.3.

2.5: Output the best chromosome(s) found.

It is noteworthy that the terminal set is dynamically extended by a subset of references to the reuseable genes. Each reference is given a unique terminal name. The reuse is done such that an upstream gene within a chromosome can only reference downstream genes within the same chromosome i.e. a gene cannot reference later (upstream) genes or itself. As in MEP, any chromosome in the population may only contain references to the preceding genes as discussed in the previous example. With this constraint it is ensured that no loops are created. Only five selected genetic operators were used for the purpose of this dissertation i.e. **Mutation**, **Root transposition within genes**, **One- and Two-point recombination** and **Gene recombination within chromosomes**. During mutation, any terminal or reference terminal in the head of the selected gene can only be mutated into a function or another terminal or reference terminal pointing to an earlier gene. Any terminal or reference terminal in the tail of the selected gene can only be mutated into another terminal or reference terminal pointing to an earlier gene. This constraint is necessary to avoid forward referencing. It should be noted that the reused genes are selected at random during the creation of genes and therefore it is possible for a particular gene not to reference any of the downstream genes at all.

The effect of encapsulation or referenced gene (sub-structure) is that the referenced gene in the newly created gene is no longer subject to the potentially disruptive effects of the crossover operator because it is now an indivisible single point. In effect, the encapsulated genes are potential building blocks for future generations and for solving the problem at hand which could be complex in nature. Note that they may proliferate in the population in later generations [15, p. 112], [7]. As discussed in Section 2.1, no function may appear in the tail of a gene in the traditional GEP. However, in IGEP, the gene reference may appear in the tail of the gene while keeping the structure of the traditional GEP intact. This feature provides an added benefit to IGEP as symbols in the tail are implicitly allowed to represent functions. The main benefits of IGEP are: gene-reuse capability and that, through encapsulation, good genes are preserved from destructive nature of the crossover operator. On the other

hand, with IGEP block structures may be obscured and the complexity of a chromosome may be concealed as a result of encapsulation. Furthermore, from an implementation point of view, as the number of reused genes increases the performance of the IGEP algorithm slows down as a result of the increasing number of terminals to be used in the computation. As an example, consider the following three-genic chromosome in the population:

Gene 1									**Gene 2**									**Gene 3**								
0	1	2	3	4	5	6	7	8	0	1	2	3	4	5	6	7	8	0	1	2	3	4	5	6	7	8
a	&	|	c	a	b	b	a	b	|	b	a	A	c	a	b	A	c	b	|	A	a	b	B	B	b	a

The new terminals, A and B in the above genes reference genes in positions 1 and 2, respectively, within the chromosome. Using this representation, the maximum number of genes that can potentially be reused or referenced is 26, using alphabets A to Z corresponding to the genes in positions 1 to 26 in the chromosome. The maximum number of 26 referenced genes is considered more than enough taking into account the complexity of the problems considered in this study. Note that in this representation, the genes are not linked, except through referencing, and each gene within a chromosome represents a potential solution to the problem as in MEP. However, in this study the best gene in terms of fitness was chosen to represent the potential solution in which the less fit genes are allowed to act as building blocks of the best gene through referencing. Executing this process, genes are reused as building blocks for individual chromosomes in a MEP fashion. It is this ability of MEP to repeatedly use the same gene (sub-structure) in an expression without repeating it that this research aims to exploit to improve efficiency of the IGEP as it is expected to take shorter time to evaluate chromosomes; since reused genes are evaluated only once and the results are reused in the subsequent stages of evaluation.

2.4 Fitness function

Fitness is the driving force of Darwinian natural selection and, likewise, of both conventional genetic algorithms and genetic programming. In nature, the fitness of an individual is the probability that it survives to the age of reproduction and reproduces. The fact that individuals exist and survive in the population and successfully reproduce may be indicative of their fitness as is the case in nature. Fitness may be measured in various ways, some explicit and some implicit. The most common approach to measuring fitness is to create an explicit fitness measure for each individual in the population [15]. The goal of this study is to produce a fully functional design (i.e. one that produces the expected behaviour stated by its truth table) which minimises the number of functions used. On this note, the calculation of fitness function F in this study is divided into two parts f_1 and f_2 that measure the functionality (i.e. compliance with the truth table) and the complexity (i.e. total number of functions used), respectively [12, 25].

Let x_i be the bit string for each truth table row and y_i the expected output. Let g_i be the value computed by the gene for input x_i, such that:

$$f_1 = \sum_{j=1}^{r} XOR(y_i, g_i), \tag{2.5}$$

f_1 represents the total number of outputs produced by the IGEP circuit not matching with the expected values, according to the truth table (on a bit-per-bit basis). The fitness value of an individual gene (G_i) is a weighted

sum of the number of incorrect results (f_1) and the number of gates used (f_2) given by:

$$f_{G_i} = w_r \times f_1 + w_g \times f_2, \tag{2.6}$$

where $w_r, w_g > 0$ are "weights" indicating the relative importance of optimality versus correctness. The sum is weighted in order to give preference to correct solutions over short expressions. In this manner, the algorithm searches for a solution that gives a correct output for each given combination of inputs and is also small (i.e. using the lowest possible number of logic gates) [13, p. 88]. In turn the fitness of each chromosome in the population was taken as $F = \min\{f_{G_1}, f_{G_2}, f_{G_3}, \ldots, f_{G_n}\}$, where n is the number of genes in the chromosome. Here, a lower "fitness" value F indicates a fitter chromosome. Clearly, when using this fitness function individuals within the population are rewarded on the basis of optimality and functionality. For the purpose of this dissertation, a number of constants were tested and; 1000 and 1 were found to be suitable for w_r and w_g, respectively. This clearly puts more emphasis on the correctness than the optimality of the chromosome.

2.5 Termination criterion

Termination criterion is a criterion by which the algorithm decides whether to continue searching or stop the search. There is a wide range of termination criteria one can choose from. These criteria can be summarised as follows [29, p. 59]:

- Maximum generations - The algorithm stops when the specified number of generations have evolved.
- Elapsed time - The genetic process will end when a specified time has lapsed.
- No change in fitness - The genetic process will end if there is no change to the population's best fitness for a specified number of iterations.

The termination criterion in this study was specified in terms of maximum number of generations. However, it may not always be possible for the algorithm to converge to exactly one individual (solution) since there may be more than one competing individuals with the same fitness, in which case there is no basis for choosing one individual over another (we may choose arbitrarily).

Chapter 3

Review of literature

The problem of evolving digital circuits has been intensely analysed in the recent past. The focus has been largely on evolving efficient or inexpensive digital circuits with fewer gates and fewer gate inputs per gate [14]. In [15] combinational circuits were designed using GP. Koza has designed, for example, a two-bit adder, using a small set of gates (AND, OR, NOT), but his emphasis has been on generating functional circuits rather than optimising them [5].

Most recently, Deng, He and Huang [6] have developed a new variant of the GP algorithm called Multi-expression based Gene Expression Programming (MGEP). As the name suggests, the MGEP is a modified GEP algorithm based on MEP. This means that the MGEP follows exactly the same evolutionary process as the traditional GEP [6]. The MGEP addresses the weaknesses of both GEP and MEP as highlighted in detail in [6]. MGEP achieves this by combining the strengths of GEP and MEP i.e the simplicity of encoding method from GEP and the ability of MEP to store multiple solutions of a problem in a single chromosome. The gene representation in MGEP is the same as in the traditional GEP i.e. the head contains symbols that represent both functions and terminals while the tail contains only terminals. As in MEP, a gene in MGEP can be decomposed into multiple expressions representing multiple solutions to a problem and the gene fitness is taken to be the fitness value of the best sub-expression. Using the example presented in Section 2.1, in MGEP, the K-expression |**&bc&a&bc** is decomposed into four ETs. The first ET is obtained by reading the K-expression from the first head character, which is exactly the same as with the traditional GEP while second ET is then obtained by reading the K-expression from the second character head and so on, see Figure 3.1:

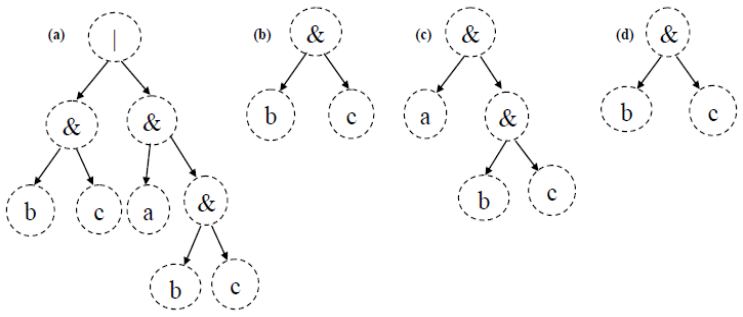

Figure 3.1: A K-expression translates into three unique sub-expressions. (a) E_1 = (b & c) | [a & (b & c)], (b) E_2 = b & c, (c) E_3 = a & (b & c), (d) E_4 = b & c

Note that unlike MEP, the MGEP algorithm does not generate an ET for a single node and it allows explicit repitition of sub-expressions, see Figure 3.1. The MGEP algorithm uses exactly the same genetic operators as the standard GEP as discussed in Section 2.1.2. Based on the experimental results of the two symbolic regression problems presented in [6], the MGEP algorithm was found to have significantly improved the evolutionary performance when compared with both the MEP and the GEP algorithms owing to the improved way of decoding and assigning of fitness to a gene in MGEP [6].

Julian Miller, one of the pioneers in the field of evolvable digital circuits, used a special technique called Cartesian Genetic Programming (CGP) [19] for evolving digital circuits. CGP was invented by Miller in 1999 and was developed from a representation of electronic circuits devised by Miller and Thomson a few years earlier. CGP is a highly efficient and flexible form of GP that encodes a computer program as an acyclic directed graph whose nodes (gates) are organised in n_c columns and n_r rows. Depending on a particular application, the nodes can be elementary logic functions, transistors or high-level components such as adders or multipliers [4, 28]. CGP represents computational structures (mathematical equations, circuits, computer programs etc.) as a string of integers. These integers, known as genes determine the functions of nodes in the graph, the connections between nodes, the connections to inputs and the locations in the graph where outputs are taken from. As discussed in [4], originally CGP used a program topology defined by a rectangular grid of nodes with a user-defined number of rows and columns. However, later work on CGP showed that it was more effective when the number rows was chosen to be one. Using a graph representation is very flexible as many computational structures can be represented as a graph. The results have shown that CGP was able to evolve digital circuits better than those designed by human experts [23]. In the context of the CGP, the chromosome shown in Figure 2.1 is represented as a list of integers as follows:

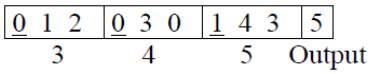

Figure 3.2: An example of CGP representation as a list of integers

In CGP, the integers are mapped to acyclic directed graphs rather than trees. One motivation for the CGP is that it uses graphs that are more general than trees. Figure 3.3 gives a graphical representation of the above chromosome. Note that 0 and 1 refer to the logic operations AND (&) and OR (|).

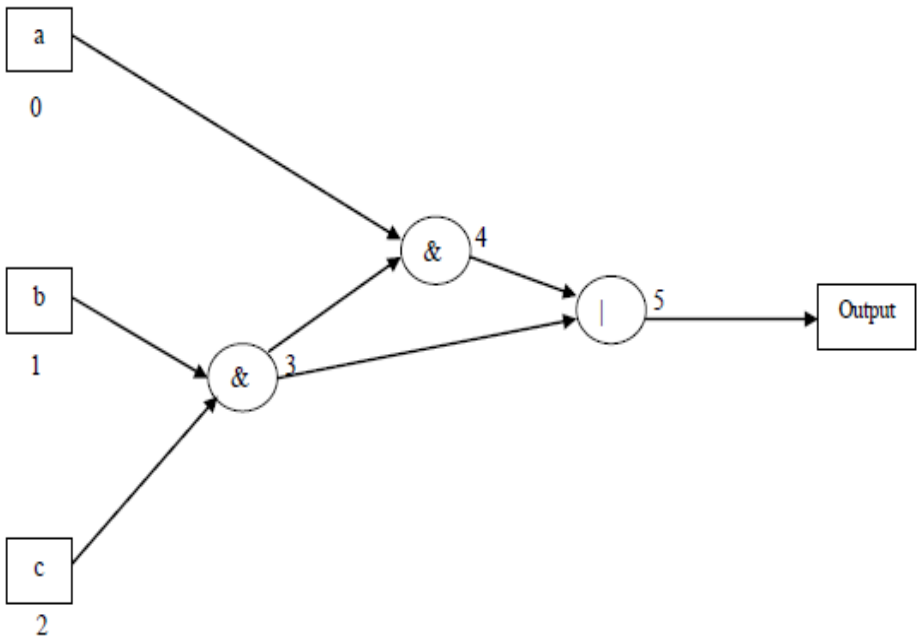

Figure 3.3: An example of CGP representation in a form of an acyclic directed graph

As correctly pointed out in [4] the benefit of this type of representation is that it allows the implicit reuse of nodes in the directed graph.

In the quest to improve the performance of Genetic Programming (GP), Koza introduced code-reuse in GP by using ADFs [15]. According to Koza *et al.* [16], an ADF is a function that is dynamically evolved during a run of genetic programming and that may be called by a calling program (or subprogram) that is concurrently being evolved. When ADFs are used, a program in the population consists of hierarchy of one (or more) reusable function-defining branches (i.e. ADFs) along with a main result-producing branch [16, 15]. As pointed by Qureshi [24, p. 24], from an implementation point of view, the introduction of ADFs adds an extra step to the preparatory steps from the GP application. The extra step defines the architecture in terms of number of ADFs and the arguments that they take, and the result-producing branch. Nonetheless, owing to the code-reuse capability of the ADFs, the use of the ADFs has been found experimentally to greatly reduce the computational effort required to generate correct computer programs [2].

On the other hand, Oltean [22, 23] conducted a comparative study between CGP and MEP. The comparison was done on the basis of computation effort spent by CGP and MEP on two problems that are well-known benchmark instances used for assessing the performance of the algorithms evolving circuits i.e. two-bit multiplier and two-bit adder with carry. The results of the numerical experiments show that MEP outperforms CGP on some of the considered test problems. In some cases the MEP was found to perform better than CGP with more than one order of magnitude. Furthermore, Oltean [23, p. 7] draws attention to some significant differences between MEP and CGP. The differences show a significant advantage to the MEP over the CGP. A detailed discussion on strengths and weaknesses of several linear genetic programming techniques, including CGP, MEP and GEP can be found in [22].

An improved version of MEP i.e. IMEP was introduced in [11]. The improvement involved rearranging the nodes of the original MEP representation. To improve efficiency, in IMEP all terminals were kept in the first positions (genes) and no other genes containing terminals were allowed in the rest of the chromosome. The IMEP representation of the chromosome shown in Figure 2.1 would then be as follows:

0: b
1: c
2: a
3: & 0, 1
4: & 2, 3
5: | 3, 4

Figure 3.4: Prefix IMEP representation of the chromosome in Figure 2.1

In addition to this new representation, another mutation function was added to allow the replacement of the worst individual in the population with the worst mutated individual, reason being that sometimes the worst individual may contain good genes to be exploited, so by mutating this individual its fitness may improve and therefore it will have a better chance to be selected. On the basis of the experiments performed, the IMEP outperformed the MEP and CGP [11].

Yan, Wei, Liang, Hu and Yao [33] implemented an improved GEP for electronic circuits using CGP. A representation was adopted in which a chromosome was represented as an n x m geometry of uncommitted logic cells with inputs, outputs and netlist numbering of integers that are mapped to directed graphs rather than trees. Based on the outcome of case studies conducted (one-bit full adder, two-bit half adder and two-bit full adder), this version of GEP produced optimum circuits and, most importantly, guaranteed the populations diversity and hence causing the population not to trap into the local optima.

Furthermore, a new technique for evolutionary design of digital circuits by way of GP with subtree mutation was proposed in [3]. Subtree mutation replaces a randomly selected subtree with another randomly created subtree [15, p. 106]. In this technique a mutation point is chosen and the subtree connected to that point is removed and it is then replaced with a newly generated subtree. The proposed technique, helps to simplify and speed up the process of designing digital circuits, discovers a variation in the field of digital circuit design where optimised digital circuits can be successfully and effectively designed [3]. The results obtained using this technique demonstrate the potential capability of genetic programming in digital circuit design with limited computer programs.

AL-Saati and AL-Assady performed a thorough assessment of GEP in [1]. A number of some inherent weaknesses with GEP in its original form as proposed by Ferreira were identified and potential solutions were proposed.

The weaknesses identified include the problem of:

a. choosing the best parameter settings,

b. using only one linking function,

c. gene flattening,

d. illegal operations in genes and

e. lack of function biasing.

The first two drawbacks were successfully addressed by applying the so called multi-population feature to the GEP. The feature involves the creation of numerous populations (P) of a given size (S) with G number of generations. The use of multi-population features enables the GEP to use different settings for each population and can therefore reduce the parameter-setting problem. Similarly, each population has its own local linking function.

The problem of flat genes is avoided by imposing some monitoring process on the application of the IS operator, so that, when the number of functions in the head is zero, an emergency mutation is forced after that IS operator to ensure that the existence of a function in the head of the modified gene. Similarly, the illegal operations in genes are avoided by adding an emergency mutation in the fitness calculation such that when an invalid operation is about to abort the fitness calculation, the operation is simply mutated to one of the remaining functions in the function set. The case studies conducted in [1] found that, to the extent possible, all these improvements enhanced the rate of successful runs.

Chapter 4

Results

Both IGEP and the standard GEP algorithms were implemented to allow comparisons of the results for some of the known problems in the area of evolutionary algorithms. The results were compared with the aim of evaluating the performance of the enhanced algorithm i.e. IGEP. Both algorithms were applied to the **fundamental logic expressions** (AND, OR, NOT, NAND, XOR and NOR), **one-bit HA**, **AND-OR ALU** and **one-bit FA** using the NAND gate. As discussed in [17, p. 85] the NAND gate possesses a special property: it is universal. That is, given enough gates, it is able to mimic the operation of any other gate type. For example, it is possible to build a circuit exhibiting the OR function using three interconnected NAND gates. The ability for a single gate type to be able to mimic any other gate type is one enjoyed only by the NAND and the NOR. This property of NAND and NOR gates is very important because, from a manufacturing point of view, it is a lot cheaper in practice to manufacture a large number of similar gates than a large number of different gates. It is for this reason that digital control systems have been designed around nothing but either NAND or NOR gates, all the necessary logic functions being derived from collections of interconnected NANDs or NORs [17, p. 85].

The comparison of the results from GEP and IGEP was done on the basis of the *fitness* (i.e. optimality and functionality) of the final logic circuits produced as well as the *success rates* of the two algoritms. The success rate is defined as the proportion of trials or runs in which the termination criterion is met [34]. In line with this definition, the success rate in this dissertation is defined as the proportion of the number of correct solutions obtained to the total number of runs.

As mentioned the NAND gate is universal and the advantage of this property was discussed. As proof of the universality property, in this section it is shown how all the basic gate types, the HA, the AND-OR ALU and the FA were formed using only NAND gates for two reasons:

- To investigate the potential efficiency of using one type of logic gate by exploiting the universality property of the NAND gate.
- To test the efficiency of the IGEP program.

Table 4.1 below shows the parameters used in testing and comparing the performance of GEP and IGEP when applied on fundamental logic expressions, HA, AND-OR ALU and FA using the NAND gate. The symbol "+" in the examples below denotes the NAND operation.

Table 4.1: GEP and IGEP parameters

Head length	8
Number of genes	4
Root transposition of IS elements rate	0.10
Two-point recombination rate	0.23
One-point recombination rate	0.70
Gene recombination rate	0.23
Mutation rate	0.05
Selection mechanism	Roulette wheel selection
Population size	100
Number of generations	500
Number of runs	50

4.1 Example 1: Designing logic circuits for basic logic gates

Table 4.2: Truth table for AND, OR, NAND, NOR, XOR and NOT

Inputs		**Outputs**					
a	b	AND(a,b)	OR(a,b)	NAND(a,b)	NOR(a,b)	XOR(a,b)	NOT(a)
0	0	0	0	1	1	0	1
1	0	0	1	1	0	1	0
0	1	0	1	1	0	1	
1	1	1	1	0	0	0	

To exploit the universality property of the NAND gate, all the basic logic gates were implemented using the NAND gate only. Tables 4.3 and 4.4 below, show the results for all the basic logic functions obtained using IGEP and the standard GEP, respectively. Note that the bold gene represents the solution.

Table 4.3: Results for basic logic gates obtained using IGEP

Logic gate	**Fitness**	**Success rate (%)**	**Prefix Logic Expression**			
			A	**B**	**C**	**D**
0	0	100	11+10001000100111	**010AA1A+1101A0AA1**	10B1+01+BB11000AA	C0BA0AB0CCB1CA0C0
1	0	100	010+1+++110011101	+1110010A10A1AAAA	A01+001+BABABB0AB	**100A1CBBBA0B1A10A**
a	0	100	01a11+0a0a01010a0	**aa01A+aAa1aAa0a0a**	B0a011a+ABBa1a0a0	+C1aAaaA0AaAAB10A
b	0	100	**bbbbbb1+bb00b10bb**	b1bA+0+0111bb001b	+b+b0++bbbbABbAAA	0C010b1+0ABAA1101
NOT	1	100	+01++++00a0001a01	**+aa1a1+AAA1AaA0AA**	B+a0A+A0aBAaaA00B	1Ba01aC1B11BCBa0A
NAND	1	100	**+ab++bbaa0b10bb0b**	+10101aAab0aAA0ab	BAB+0+aaBaA1bAbbA	aA0C00+C0bbaaACA1
AND	2	100	+ab1+1+110b0b1010	**++ab1+1aabbabba0a**	bb00Ba0B101bbBbBa	bAbaB1bAbbAa1bbAA
OR	3	100	+a10aaa010b1b0010	**+A+1b11aa0a1Aa0Aa**	+bBAbA0Aab1Bb01BB	+aA1+b11b11CABAAC
XOR	5	4	+ba00bbbaab1b0abb	**+++b1a+bAA1ab00aa**	++B+b1+10b010aaBB	+AAC01+AAabA0BB00
NOR	4	34	+aaaaa1+ba0bbb11a	1aA0bAa+baa0b1Aaa	**+B+A+BbAB1abaBA0a**	++++0bb+Ba1ababA1

Table 4.4: Results for basic logic gates obtained using GEP

Logic gate	**Fitness**	**Success rate (%)**	**Prefix Logic Expression**			
			A	**B**	**C**	**D**
0	0	100	+1101+1+110110010	+++10000101111010	100+00+1110111000	**0+1+1001010010011**
1	0	100	**1101+111010000100**	11++1011110010000	010010++011010000	+1+1+++0101011011
a	0	100	10a0++0a00a11aaaa	+1a0a+0+01aa0010a	**a0a010a1a001aa111**	+0+1110a11aa01aaa
b	0	100	00b0b+bb1bb1b00bb	**b+0+0bbb0bb001bbb**	b00b0b1bb00b00bbb	b11b01++00b1b11bb
NOT	1	100	**+1a0101aa11aa0aa0**	+11a1a1a0011a011a	0+01111aa101011a0	++0aa00aa1a000aa0
NAND	1	100	+++101aa01b0a11a1	**+ab000+b100b0b0a1**	ab11b0b+11bba10b0	+b11ba0b01baa0ba1
AND	2	100	**++ab110ab100aa0b0**	bb0baaa111b0b011a	+a++++++110babb00	++++++++0b111bbaa
OR	4	98	**++b+0b+aa0a1a111b**	++++++++a1b1b1b1b	++++++++001bb1b01	++++++++0111bba11
XOR	5	2	**++b+1a++baab00aaa**	b1ba101b1a0aab1aa	++11+1+1aa1b1a10b	++++1a++baab00aaa
NOR	5	32	**++b+a1++aba1baa01**	+ab10a1abaaa00b11	0bb+a1++aba1baa01	1aab1a1b01a000ba1

Comparing Tables 4.3 and 4.4 in terms of fitness (i.e. the same fitness function is used for both methods), it is clear that for simple problems the performance of IGEP is comparable to that of the standard GEP. However, for a slightly more complex problems i.e. OR and NOR, IGEP yielded a better solutions compared to GEP. Furthermore, the results show that the success rate of IGEP is about 2% more compared to GEP for the OR, NOR and XNOR logic functions. This is due to the extensive gene-reuse in IGEP as evident in the results. As expected, the level of gene-reuse in IGEP increases with the complexity of the circuit being evolved, see the bold genes in Table 4.3.

4.2 Example 2: Logic circuits for an AND-OR ALU and one-bit HA

In this example, the IGEP was used to derive the logic expressions for the AND-OR ALU and HA. We used 4 genes, each with a head length of 8. As in the previous example only the NAND gate was used. The AND-OR ALU performs the AND and OR logic functions. The design has two inputs, a and b, and one select bit (c) to select between the two functions, AND and OR.

Table 4.5: Truth table for an AND-OR ALU

Inputs			**Outputs**
a	b	c	d
0	0	1	0
0	1	1	0
1	0	1	0
1	1	1	1
0	0	0	0
0	1	0	1
1	0	0	1
1	1	0	1

The HA adds two single binary digits, a and b. It has two outputs, sum (c) and carry (d). The carry signal represents an overflow into the next digit of a multi-digit addition.

Table 4.6: Truth table for one-bit HA

Inputs		**Outputs**	
a	b	c	d
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

The results presented in Tables 4.7 and 4.8 suggest that in terms of fitness, for more complex problems i.e. AND-OR ALU and HA, IGEP yielded better solutions compared to GEP with the exception of the SUM part of the HA. In fact GEP could not solve the AND-OR ALU. Again, the IGEP was found to be more successful in finding correct solutions than GEP, specifically for the AND-OR ALU and HA(Carry). These results prove that the performance of IGEP has been enhanced through gene-reuse.

Table 4.7: Results for an AND-OR ALU and one-bit HA obtained using IGEP

Logic gate	**Fitness**	**Success rate (%)**	**Prefix Logic Expression**			
			A	**B**	**C**	**D**
AND-OR ALU	5	4	cbb1cac0acb11abcc	++bac1+1010b1cc10	**++bB+BaAaBABBa10B**	+BcA0BbB1C1b0cB0A
HA (Sum)	2	100	+abbbababbaa1bb11	**+A11+aa+0a0b11bbb**	B1a01bb+babbab0ab	0A00B0+aC0b1AbaC0
HA (Carry)	4	12	+bab+1+aaba0b0ba0	**++Ab+aA+A0b000010**	+0000Aa+0A0A1aBBB	b0+1+a+1BA1aA0aaa

Table 4.8: Results for an AND-OR ALU and one-bit HA obtained using GEP

Logic gate	**Fitness**	**Success rate (%)**	**Prefix Logic Expression**			
			A	**B**	**C**	**D**
AND-OR ALU	1 002	0	**+c+baccb11aaa00bc**	+cbbaccb11aaa00bc	+++++++01a11baa1a	ab1bcabbab1abb1bc
HA (Sum)	2	100	**++ba11+babb0110b1**	++++++++101b00b0a	++++++++a000b10bb	++++++++b100001aa
HA (Carry)	5	4	**+++abb++baa00abaa**	ba0+10bbb1bb101b1	+ab0b+b110aab01b0	b+1a0+a1a100aaa00

4.3 Example 3: Logic circuits for one-bit FA

The FA circuit adds three one-bit binary numbers (a b c) and outputs two one-bit binary numbers, a sum (d) and a carry (e). As in the previous examples, the circuit was derived using the NAND gate only.

Table 4.9: Truth table for one-bit FA

Inputs			**Outputs**	
a	b	c	d	e
0	0	0	0	0
1	0	0	0	1
0	1	0	0	1
1	1	0	1	0
0	0	1	0	1
1	0	1	1	0
0	1	1	1	0
1	1	1	1	1

The optimum logic expression for the SUM part of the FA was obtained using IGEP owing to the extensive gene-reuse as observed in Table 4.10. In this particular instance the IGEP was not successful in finding the solution for the CARRY part of the FA while on the other hand the standard GEP algorithm failed to solve both the SUM and CARRY of the FA as shown in Table 4.11.

Table 4.10: Results for one-bit FA obtained using IGEP

Logic gate	**Fitness**	**Success rate (%)**	**Prefix Logic Expression**			
			A	**B**	**C**	**D**
FA (Sum)	6	4	+1a+a10+ab11a01c1	++bcA110bbAcccb0c	**++bB+acbc00AAB0cb**	+aa+0c1bbbA0B0B01
FA (Carry)	2 006	0	++acb00+10cbb1111	+c+1b++b0Aacba0cb	cc1ccB1bcBab1bccA	**+B+aAa1cab0ccA0bB**

Table 4.11: Results for one-bit FA obtained using GEP

Logic gate	**Fitness**	**Success rate (%)**	**Prefix Logic Expression**			
			A	**B**	**C**	**D**
FA (Sum)	2 000	0	**cc0b0a0110ac1aaab**	1ab+a++0c01aac0b1	ac100abaac0cc0bc1	+0+ca0c0cbc1c0cb0
FA (Carry)	3 002	0	**+a+bc++b00110ba01**	+b+bc++b00110ba01	+c+++ccca1cacaabc	cc01aa00c1cb0b011

Chapter 5

Conclusion

In this dissertation, two forms of genetic programming, GEP and MEP techniques, were explored in detail in order to demonstrate and understand the two techniques with the aim of integrating them. In line with the objectives of the study, this research has successfully integrated the features of MEP into the standard GEP algorithm to improve the performance and efficiency of GEP, notably the code-reuse functionality of MEP. As in MEP, the code-reuse in the IGEP was achieved through a representation which allows multiple references to a single sub-structure.

To assess the performance of IGEP, the IGEP algorithm was applied on nine known problems in the area of circuit design, ranging from simple to more complex problems i.e. AND, NAND, NOT, OR, XOR, NOR, AND-OR ALU, one-bit HA and one-bit FA. The performance of IGEP was compared with that of the standard GEP. These two techniques produced comparable results for simple problems such as basic logic functions. However, for complex problems such as one-bit HA(CARRY), FA(SUM) and AND-OR ALU the IGEP performed better than the standard GEP. Furthermore, the success rate of IGEP was found to be generally higher than that of GEP due to the gene-reuse capability implemented in IGEP.

As part of future work, I suggest we further explore and improve the robustness of IGEP such that it can solve any problem, even more complex problems such as the one-bit FA(CARRY). Furthermore, it would also be useful to develop an estimate for the optimum head length needed to successfully implement a given a logic circuit.

Appendix A

Implementation of the IGEP algorithm

The C++ program given in A.2 implements the IGEP. The program takes an input file and produces two output files. One can switch between IGEP and GEP by specifying "igep" or "gep", respectively, when asked to choose between the two algorithms. The input and output files are described in Section A.1 below.

A.1 Input and output description

Table A.1 below summarises the structure of the input file by providing a description of what each line in the input file represents.

Table A.1: Summary of the input file

Line	Description
1	Number of states (rows) of the truth table
2	Number of columns (number of inputs and outputs) of the truth table
3	Number of inputs (terminals), excluding 0 and 1
4	Number of outputs of the truth table
5	Number of inputs (terminals), including 0 and 1
6	List of inputs (terminals), including 0 and 1
7	The seventh row onwards gives the truth table

As an example, the input file for the "AND" logic operator would look as follows:

```
4
3
2
1
4
ab01
000
100
010
111
```

Table A.2 below shows the list of logic operations considered in this dissertation. However, it should be noted that the results presented in this dissertation were produced using the "NAND" operation only but this program, with minor modification, can be executed using any operation or a combination of operations given in Table A.2.

Table A.2: List of operations

Symbol	Operation (Logic gate)
+	NAND
&	AND
!	NOT
|	OR
*	XOR
$	NOR

There are two output files. One of these files is the main output file which contains the fittest chromosome in the population for each run. The file also gives, for each fittest chromosome, the fitness value as well as the iteration on which the chromosome was generated. Also provided in the file is the position of the gene which represents the solution and an indication of whether or not the algorithm was successful in finding the solution. The second output file contains the average fitness for each run.

A.2 The IGEP algorithm

```
1  #include <sstream>
2  #include <cstdio>
3  #include <string>
4  #include <cstring>
5  #include <iostream>
6  #include <fstream>
7  #include <stack>
8  #include <vector>
9  #include <cmath>
10 #include <algorithm>
11 #include <ctime>
12 using namespace std;
13 /*------------------------------------------------------------------------------*/
14 ofstream outputFile;//define the output file (1st & last iterations)
15 ofstream oFile;//define the output file for avg fitness
16 string filename_out;//define the output file
17 //define control parameters
18 int head_length = 8,
19     num_genes = 4,
20     tail_length,
```

```
21      gene_length,
22      outputs,//number of outputs
23      popsize = 100,
24      max_generations = 500,
25      num_runs = 50,
26      num_replicates = popsize*0.1,//10% of the population, including the cloning of best
            chromosome
27      num_terminals,//specify number of terminals
28      num_functions, //number of functions
29      table_rows,//specify number of rows for truth table
30      table_cols,//specify number of columns for truth table
31      first_output_col,//specify the input column
32      num_inputs,//specify number of inputs
33      fs_ter_gr_len,
34      gen_count;
35  bool *truth_table_pointer;//pointer to the truth table
36  bool mutate = true;
37  bool elitism = true;
38  double *avg_fitness;
39  string functions = "+" /*"!+*&|$"*/, //+->NAND &->AND !->NOT |->OR *->XOR $->NOR
40         ref_genes = "ABCD",//referenced genes
41         fs_ter_gr,
42         terms,
43         reuse;//this variable is used for switching between the Std GEP and GEP with reuse
               capability
44              //reuse = igep => invoke IGEP, otherwise invoke the Std GEP.
45  //=============================================================================
46  //test if the input string is a function
47  bool isFunction(const char &f){
48      for (int i = 0; i < num_functions; i++){
49          if (functions[i] == f) {
50              return true;
51          }
52      }
53      return false;
54  }
55  //=====================================================================
56  //random number generator (returns an integer between 0 and n-1
57  int rand_int (const int &n){
58      return int(double(n)*rand()/RAND_MAX)%n;
59  }
60  //=====================================================================
```

```
61 //random number generator (returns a float random between 0 and 1
62 double randomOne(){
63       return rand()/((double)RAND_MAX + 1);
64 }
65 //=========================================================================
66 //define a NAND function
67 bool nand(const bool &b1, const bool &b2){
68     return !(b1 && b2);
69 }
70 //=========================================================================
71 //define a NOR function
72 bool nor(const bool &b1, const bool &b2){
73     return !(b1 | b2);
74 }
75 //=========================================================================
76 //define an XOR function
77 bool xorr(const bool b1, const bool b2){
78     return !(b1 == b2);
79 }
80 //=========================================================================
81 //function to convert a CHAR to STRING
82 string Char_to_String (const char &ch){
83    stringstream str1;
84    string str;
85    str1 << ch;
86    str1 >> str;
87    return str;
88 }
89 //=========================================================================
90 //this function creates a gene.
91 string create_Gene_String(int gene_counter){
92    string gene;
93    int i = 0;
94    if (reuse != "igep"){
95        gene_counter = 0;
96    }
97    for (int h = 0;h <head_length; h++){//head
98        gene.append(Char_to_String(fs_ter_gr[rand_int(num_functions+num_terminals+
              gene_counter)]));//head
99    }
100   for (int t = 0;t <tail_length; t++){//tail
101       gene.append(Char_to_String(fs_ter_gr[num_functions+rand_int(num_terminals+
```

```
            gene_counter)]));//head
102     }
103     return gene;
104 }
105 //=====================================================================
106 //this function maps a CHAR to an INTEGER.
107 int backref_to_int(const char &x){
108     return x - 'A';
109 }
110 //=====================================================================
111 string exp_tree;//store the expression tree in a string
112 int s;//string index
113 //=====================================================================
114 //this function creates an expression tree based on a string
115 void create_exp_helper(const string &str, int str_len){
116     s++;
117     char symbol = str[s];
118     if (isFunction(symbol)){
119         if(symbol == '!'){
120            exp_tree.append(Char_to_String(symbol));
121            create_exp_helper(str,str_len);
122         }
123         else {
124             exp_tree.append(Char_to_String(symbol));
125             create_exp_helper(str,str_len);
126             create_exp_helper(str,str_len);
127         }
128     }
129      else {
130           exp_tree.append(Char_to_String(symbol));
131     }
132 }
133 //=====================================================================
134 //this function creates an expression tree given a string
135 string create_expr_tree(const string &str){
136     s = -1; //reset the string index
137     exp_tree.clear();//reset the global vector: exp_tree
138     create_exp_helper(str,str.size());//create the expression tree from a string
139     return exp_tree;
140 }
141 //=====================================================================
142 vector<vector<bool> > gene_values;
```

```
143 vector<bool> chrom_values;
144 vector<bool> v;
145 //=====================================================================
146 //this function evaluates expression tree
147 bool evaluate_ET (const string &expression,const int &row){
148     stack<bool> s;
149     bool x, y, n;// a boolean to be pushed on the stack
150     bool result;//a result after performing logic operation
151     string exp = expression;
152     reverse(exp.begin(),exp.end());
153     const int &len = exp.size();
154     for(unsigned int i = 0;i < len; i++){
155         char c = exp[i];
156         switch(c){
157                 case '0': s.push(false);break;
158                 case '1': s.push(true);break;
159                 case 'a': s.push(*(truth_table_pointer + table_cols * row + 0));break;
160                 case 'b': s.push(*(truth_table_pointer + table_cols * row + 1));break;
161                 case 'c': s.push(*(truth_table_pointer + table_cols * row + 2));break;
162                 case 'd': s.push(*(truth_table_pointer + table_cols * row + 3));break;
163                 case 'e': s.push(*(truth_table_pointer + table_cols * row + 4));break;
164                 case 'f': s.push(*(truth_table_pointer + table_cols * row + 5));break;
165                 case 'A': s.push(v[backref_to_int('A')]);break;
166                 case 'B': s.push(v[backref_to_int('B')]);break;
167                 case 'C': s.push(v[backref_to_int('C')]);break;
168                 case 'D': s.push(v[backref_to_int('D')]);break;
169                 case 'E': s.push(v[backref_to_int('E')]);break;
170                 case 'F': s.push(v[backref_to_int('F')]);break;
171                 case 'G': s.push(v[backref_to_int('G')]);break;
172                 case 'H': s.push(v[backref_to_int('H')]);break;
173                 case 'I': s.push(v[backref_to_int('I')]);break;
174                 case 'J': s.push(v[backref_to_int('J')]);break;
175                 case 'K': s.push(v[backref_to_int('K')]);break;
176                 case 'L': s.push(v[backref_to_int('L')]);break;
177                 case 'M': s.push(v[backref_to_int('M')]);break;
178                 case 'N': s.push(v[backref_to_int('N')]);break;
179                 case 'O': s.push(v[backref_to_int('O')]);break;
180                 case 'P': s.push(v[backref_to_int('P')]);break;
181                 case 'Q': s.push(v[backref_to_int('Q')]);break;
182                 case 'R': s.push(v[backref_to_int('R')]);break;
183                 case 'S': s.push(v[backref_to_int('S')]);break;
184                 case 'T': s.push(v[backref_to_int('T')]);break;
```

```
185                 case 'U': s.push(v[backref_to_int('U')]);break;
186                 case 'V': s.push(v[backref_to_int('V')]);break;
187                 case 'W': s.push(v[backref_to_int('W')]);break;
188                 case 'X': s.push(v[backref_to_int('X')]);break;
189                 case 'Y': s.push(v[backref_to_int('Y')]);break;
190                 case 'Z': s.push(v[backref_to_int('Z')]);break;
191                 case '+': x = s.top();
192                           s.pop();
193                           y = s.top();
194                           s.pop();
195                           result = nand(x,y);
196                           s.push(result);break;
197                 case '&': x = s.top();
198                           s.pop();
199                           y = s.top();
200                           s.pop();
201                           result = x && y;
202                           s.push(result);break;
203                 case '|': x = s.top();
204                           s.pop();
205                           y = s.top();
206                           s.pop();
207                           result = x || y;
208                           s.push(result);break;
209                  case '*': x = s.top();
210                           s.pop();
211                           y = s.top();
212                           s.pop();
213                           result = xorr(x,y);
214                           s.push(result);break;
215                  case '$': x = s.top();
216                           s.pop();
217                           y = s.top();
218                           s.pop();
219                           result = nor(x,y);
220                           s.push(result);break;
221                  case '!': y = s.top();
222                           s.pop();
223                           result = !y;
224                           s.push(result);
225                  default : s.push(false);break;
226         }
```

```
227     }
228     return s.top();
229 }
230 //=====================================================================
231 //this function removes duplicates in a string
232 string remove_duplicates(string & s1){
233     int n = s1.size();
234     for (int i = n-1;i != -1;--i){
235         for (int j=0;j<i;++j){
236           if (s1[i]==s1[j]){
237               int k = i;
238               while (k != n){
239                     s1[k]=s1[k+1];
240                     k++;
241               }
242           }
243         }
244     }
245     return s1;
246 }
247 //=====================================================================
248 //maxx_element returns the largest integer in an input array - this function will
249 int index_for_max_fitness (const vector<double> &num_args, const int &len){
250     int index = 0;
251     double result = num_args.at(index);
252     for (int i = 1; i < len ;i++){
253         if (result > num_args.at(i)) {
254             index = i;
255             result = num_args.at(i);
256         }
257     }
258     return index;
259 }
260 //=====================================================================
261 //define Chromosome class
262 class Chromosome{
263 public:
264     //gene variables
265     vector<string> genes;//stores the GENES in a form of string
266     vector<string> g_expr_trees;
267     vector<double> g_fitness;
268     vector<int> g_matches;
```

```
269     vector<int> g_complex;
270     int c_generation;
271     double c_fitness;
272     int c_fitness_index;
273     int c_matches;
274     //functions
275     void setChromosome(vector<string>);
276     void printChromosome();
277     void prntChromosomeTree();
278  };
279
280  void Chromosome::setChromosome(vector<string> genes_){
281     vector<string> gETs(num_genes);
282     gene_values.clear();//ensure that there are no gene values before computations
283     genes = genes_;//initialize the genes
284     for(int j = 0; j <num_genes;j++){
285         string str = create_expr_tree(genes[j]);
286         gETs[j] = str;
287     }
288     g_expr_trees = gETs;
289     vector<string>().swap(gETs);//release memory
290     //for each state, calculate the value of each gene and store in gene_values for reuse in
          later steps
291     for(int j = 0; j<table_rows;j++){
292         v.clear();
293         for (int i = 0; i < num_genes;i++){
294              v.push_back(evaluate_ET(g_expr_trees[i],j));
295         }
296         gene_values.push_back(v);
297     }
298     //calculate fitness
299     g_fitness.clear();
300     g_complex.clear();
301     g_matches.clear();
302     for (int p = 0; p<num_genes; p++){
303          int num_non_matches = 0;
304          for (int j = 0; j < table_rows;j++){
305               vector<bool> va = gene_values[j];
306              //count the number of truth table values not matching the output
307              if (va[p] != *(truth_table_pointer + table_cols * j + first_output_col)){
308                 num_non_matches++;
309             }
```

```
310         }
311     string gexp = g_expr_trees[p];
312     int num_functions = 0;
313     for (string::iterator i = gexp.begin();i != gexp.end();++i){
314         if (isFunction(*i)){
315            num_functions++;//measures complexity
316          }
317     }
318     g_complex.push_back(num_functions);
319     string str = remove_duplicates(gexp);
320     int rr = 0;
321     for(int w = 0; w <str.size();w++){
322         int q = backref_to_int(str[w]);
323         if (q >= 0 && q < 26){
324             rr = rr + g_complex.at(q);
325         }
326     }
327     g_matches.push_back(table_rows - num_non_matches);
328     g_complex[p] = num_functions + rr;
329     //fitness = w_functions x number of functions + w_errors x non-matches
330     double w_functions = 1, w_errors = 1000;
331     g_fitness.push_back(w_functions*((double)(num_functions + rr)) + w_errors*(double)
          num_non_matches);
332     }
333     c_fitness_index = index_for_max_fitness(g_fitness,num_genes);
334     c_fitness = g_fitness[c_fitness_index];
335     c_matches = g_matches[c_fitness_index];
336     c_generation = gen_count;
337 }
338
339 void Chromosome::printChromosome(){
340      outputFile<<ref_genes[c_fitness_index]<<" : ";
341      for (int i = 0;i < num_genes;i++){
342           outputFile<<genes[i]<<" ";
343      }
344 }
345
346 void Chromosome::prntChromosomeTree(){
347      outputFile<<ref_genes[c_fitness_index]<<" : ";
348      for (int i = 0;i < num_genes;i++){
349           outputFile<<g_expr_trees[i]<<" ";
350      }
```

```
}
Chromosome *populace;//declare population of chromosome
//=====================================================================
Chromosome create_Chromosome(){
    vector<string> c_;
    for(int i = 0; i<num_genes; i++){
        c_.push_back(create_Gene_String(i));
    }
    Chromosome member;
    member.setChromosome(c_);
    vector<string>().swap(c_);
    return member;
}
//=====================================================================
//this function implements Roulette Wheel Selection method
int roulette_selection(){
    int winner;
    vector<double> probs;
    vector<double> cumulative;
    double sum = 0.0;
    for (int i = 0; i < popsize;i++){
        sum = (double)(sum + (1/(double)(populace[i].c_fitness))); //take the inverse of the
             fitness value because the individual with smallest fitness is the fittest
    }
    for (int i = 0; i < popsize;i++){
        //Again, take the inverse of the fitness value because the individual with smallest
            fitness is the fittest
        probs.push_back((double)(1/((double)populace[i].c_fitness))/sum);
    }
    for (int m = 0; m < probs.size(); m++){
        double sum_probs = 0.0;
        for (int p = 0; p <= m ; p++){
            sum_probs = (double)(sum_probs + probs.at(p));
        }
        cumulative.push_back(sum_probs);
    }
    double r1 = randomOne();
    int index1 = 0, index2 = 0;
    for (int w = 0; w < cumulative.size(); w++){
        if (cumulative.at(w) >= r1) {
            index1 = w; break;
        }
```

```
391     }
392     return index1;
393 }
394 //====================================================================
395 //this function calculates average fitness
396 double calc_avg_fitness(){
397     double sum = 0;
398     for (int i = 0; i < popsize;i++){
399           Chromosome member = populace[i];
400           sum = sum + (double)populace[i].c_fitness;
401     }
402     return sum/popsize;
403 }
404 //====================================================================
405 //this function searches for the worst chromosome & replaces it with
406 //the newly created one provided it is better than the worst
407 void replace_worst_chr(Chromosome member) {
408     Chromosome member1, member2;
409     member1 = populace[0];
410     double worst = member1.c_fitness;
411     int index = 0;
412     for (int m = 1; m < popsize; m++) {
413         if (populace[m].c_fitness > worst) {
414            worst = populace[m].c_fitness;
415            index = m;
416         }
417     }
418     if (member.c_fitness <= worst){
419         populace[index] = member;
420     }
421 }
422 //====================================================================
423 // this function finds the fittest chromosome in the population
424 Chromosome find_fittest(){
425     Chromosome fittest = populace[0];
426     for(int i = 1; i < popsize; i++){
427         if (fittest.c_fitness > populace[i].c_fitness){
428             fittest = populace[i];
429         }
430     }
431    return fittest;
432 }
```

```
433 //=====================================================================
434 //this function performs mutation
435 Chromosome Mutation(Chromosome cr){//within gene
436     vector<string> genes = cr.genes;
437     for (int g = 0; g < num_genes; g++){
438         if (randomOne() < 0.05) {//mutation rate = 0.05
439             string rg = genes[g];//choose a random gene
440             int rr, gg = g;
441             if (reuse != "igep"){
442                 gg = 0;
443             }
444             if (rand_int(gene_length) < head_length) {
445                 rr = rand_int(head_length);//pick any pos in the head of a gene
446                 rg[rr] = fs_ter_gr[rand_int(num_functions+num_terminals+gg)];
447             }
448             else{
449                  rr = head_length+rand_int(tail_length);//pick any pos in the tail of a gene
450                  rg[rr] = fs_ter_gr[num_functions+rand_int(num_terminals+gg)];
451             }
452             genes[gg] = rg;
453         }
454     }
455     cr.setChromosome(genes);
456     return cr;
457 }
458 //=====================================================================
459 //this function performs Root transposition
460 void Root_transposition(){//within gene
461     int winner1 = roulette_selection();
462     Chromosome member = populace[winner1];
463     if (randomOne() < 0.1){ //root insertion transposition rate = 0.1
464         vector<string> genes = member.genes;
465         //select a gene at random
466         int r = rand_int(num_genes);
467         string rg = genes[r];//choose a random gene
468         string head_ = rg.substr(0,head_length),
469                tail_ = rg.substr(head_length,tail_length);
470         int pos = 1 + rand_int(head_length - 1);//choose a random position in the head but
              not the first position
471         string ris_;//to store the ris element
472         int flag = 0;
473         while (pos > 0) {
```

```
474             ris_.append(Char_to_String(head_[pos]));//construct the ris element
475             if (isFunction(head_[pos])){
476                flag = 1;break;
477             }
478             pos--;
479            }
480         if (flag == 1){
481              reverse(ris_.begin(),ris_.end());
482              int ris_length = ris_.size();
483              string su_ = head_.substr(0,head_length-ris_length);
484              string du_ = ris_ + su_ + tail_;
485              genes[r] = du_;
486              member.setChromosome(genes);
487         }
488     }
489     if (mutate) {
490         member = Mutation(member);
491     }
492     replace_worst_chr(member);
493 }
494 //=====================================================================
495 //this function performs 1 point recombination
496 void OnePoint_recombination(){
497     int winner1 = roulette_selection();
498     // choose a random point in the chromosomes and exchange material
499     Chromosome member1 = populace[winner1];
500     int winner2 = roulette_selection();
501     while(winner1==winner2){
502         winner2 = roulette_selection();
503     }
504     Chromosome member2 = populace[winner2];
505     vector<string> g1 = member1.genes;
506     vector<string> g2 = member2.genes;
507
508     if (randomOne() < 0.7){//1pt recombination rate = 0.7
509         string chr1, chr2;
510         for (int j = 0;j<num_genes;j++){
511             string g_1 = g1[j], g_2 = g2[j];
512             chr1.append(g_1);
513             chr2.append(g_2);
514         }
515         int pos1 = rand_int(num_genes * gene_length),
```

```
516             pos2 = chr1.size();
517
518         swap_ranges(chr1.begin()+pos1, chr1.begin()+pos2, chr2.begin()+pos1);
519         for (int i = 0; i<num_genes;i++){
520             string genes1, genes2;
521             for(int k = gene_length*i; k < gene_length*(i+1); k++){
522                 genes1.append(Char_to_String(chr1[k]));
523                 genes2.append(Char_to_String(chr2[k]));
524             }
525             g1.at(i) = genes1;
526             g2.at(i) = genes2;
527         }
528         member1.setChromosome(g1);
529         member2.setChromosome(g2);
530     }
531     if (member1.c_fitness > member2.c_fitness){
532         member1 = member2;
533     }
534     if (mutate) {
535         member1 = Mutation(member1);
536     }
537     replace_worst_chr(member1);
538 }
539 //=====================================================================
540 //this function performs 2 point recombination
541 void TwoPoint_recombination(){
542     int winner1 = roulette_selection();
543     // choose a random point in the chromosomes and exchange material
544     Chromosome member1 = populace[winner1];
545     int winner2 = roulette_selection();
546     while(winner1==winner2){
547         winner2 = roulette_selection();
548     }
549     Chromosome member2 = populace[winner2];
550     vector<string> g1 = member1.genes;
551     vector<string> g2 = member2.genes;
552
553     if (randomOne() < 0.233333333){//2pt recombination rate = 0.233333333
554         string chr1, chr2;
555         for (int j = 0;j < num_genes; j++){
556             string g_1 = g1[j], g_2 = g2[j];
557             chr1.append(g_1);
```

```
558             chr2.append(g_2);
559         }
560         int pos1 = rand_int(num_genes * gene_length),
561             pos2 = rand_int(num_genes * gene_length);
562         if (pos1 > pos2){
563             int temp = pos2;
564             pos2 = pos1;
565             pos1 = temp;
566         }
567         swap_ranges(chr1.begin() + pos1, chr1.begin() + pos2, chr2.begin() + pos1);
568         for (int i = 0; i < num_genes; i++){
569              string genes1, genes2;
570              for(int k = gene_length*i; k < gene_length *(i+1); k++){
571                 genes1.append(Char_to_String(chr1[k]));
572                 genes2.append(Char_to_String(chr2[k]));
573             }
574             g1.at(i) = genes1;
575             g2.at(i) = genes2;
576         }
577         member1.setChromosome(g1);
578         member2.setChromosome(g2);
579     }
580         if (member1.c_fitness > member2.c_fitness){
581             member1 = member2;
582     }
583     if (mutate) {
584         member1 = Mutation(member1);
585     }
586     replace_worst_chr(member1);
587 }
588 //=====================================================================
589 //this function performs Gene recombination within a chromosome
590 void Gene_recombination(){//within chromosome
591     int winner1 = roulette_selection();
592     // choose a random point in the chromosomes and exchange material
593     Chromosome member1 = populace[winner1];
594     int winner2 = roulette_selection();
595     while(winner1==winner2){
596         winner2 = roulette_selection();
597     }
598     Chromosome member2 = populace[winner2];
599     vector<string> g1 = member1.genes;
```

```
600     vector<string> g2 = member2.genes;
601     if (randomOne() < 0.233333333){//Gene recombination rate = 0.233333333
602         int r = rand_int(num_genes);
603         string random_gene1 = g1[r],
604                random_gene2 = g2[r];
605         g1[r] = random_gene2;
606         g2[r]=random_gene1;
607         member1.setChromosome(g1);
608         member2.setChromosome(g2);
609     }
610     if (member1.c_fitness > member2.c_fitness){
611         member1 = member2;
612     }
613     if (mutate) {
614         member1 = Mutation(member1);
615     }
616     replace_worst_chr(member1);
617 }
618 //=====================================================================
619 void print_Pop(Chromosome c[], int& length){
620     for (int m = 0; m < length; m++){
621          Chromosome member = c[m];
622          member.printChromosome();
623          outputFile<<"----> "<<member.c_fitness<<" "<<member.c_generation<<endl;
624     }
625 }
626 //=====================================================================
627 // this function sort the population in ascending order w.r.t fitness
628 //this function will assist in implementing the elitism algorithm
629 void sorting_function(){
630     for(int i=0; i<popsize;i++){
631         for(int j=0;j<popsize;j++){
632             if(populace[i].c_fitness < populace[j].c_fitness){
633                Chromosome temp = populace[i];
634                populace[i]=populace[j];
635                populace[j]=temp;
636             }
637         }
638     }
639 }
640 //=====================================================================
641 //this function performs replication and elitism
```

```
642 void Replication(){
643     Chromosome *replicates;
644     replicates = new Chromosome[num_replicates];
645     for(int i = 0; i < num_replicates; i++){
646         int winner = roulette_selection();
647         replicates[i] = populace[winner];
648     }
649     if (elitism) {
650         sorting_function();//sort the population in ascending order such that the best
                chromosome are placed at the top
651         int elite = popsize*0.05;//Apply elitism - transfer the top 5% chromosomes (
                unchanged) to the next generation
652         for(int j=0;j<elite;j++){
653             replicates[j] = populace[j];
654         }
655     }
656     for(int j = 0; j < num_replicates; j++){
657         populace[j] = replicates[j];
658     }
659 }
660 //=====================================================================
661 void next_Generation (){
662     Replication();
663     int j = 0;
664     while (j < (popsize - num_replicates)){
665        if (j < (popsize - num_replicates)) {Root_transposition(); j++;};
666        if (j < (popsize - num_replicates)) {OnePoint_recombination(); j++;};
667        if (j < (popsize - num_replicates)) {TwoPoint_recombination(); j++;};
668        if (j < (popsize - num_replicates)) {Gene_recombination(); j++;};
669     }
670 }
671 //=====================================================================
672 //this function returns a minimum or maximum element in a vector
673 int min_max(vector<int> v, string maxmin){
674    if (v.size()==0){//if there is no success
675        return -1;
676    }
677    int aa = v.at(0);
678    if(maxmin == "min"){
679        for(int i=0; i<v.size();i++){
680            if (v.at(i) < aa){
681                aa = v.at(i);
```

```
682             }
683         }
684     }
685     else{
686         for(int i=0; i<v.size();i++){
687             if (v.at(i) > aa){
688                 aa = v.at(i);
689             }
690         }
691     }
692     return aa;
693 }
694 //=========================================================================
695 void createPopulation(){
696     cout<<"-------------------------------------------------------"<<endl;
697     cout<<endl;
698     cout << "Please wait while GEP generates the initial population...!"<<endl;
699     cout<<endl;
700 
701     for (first_output_col = num_inputs; first_output_col < (outputs + num_inputs);
          first_output_col++){
702         vector<int> success_iter;
703         int success_cnt = 0;
704         double fittest_per_run[num_runs][max_generations];//declare an array to save fitness
               value for best individual in every run and generation
705         for(int qq=0; qq < num_runs; qq++){
706             populace = new Chromosome[popsize];
707             avg_fitness = new double[max_generations];
708             for (int m = 0; m < popsize; m++){
709                 Chromosome member = create_Chromosome();
710                 populace[m] = member;
711             }
712             avg_fitness[0] = calc_avg_fitness();
713             int p;
714             Chromosome best_chromosome = find_fittest();
715             fittest_per_run[qq][0]=best_chromosome.c_fitness;
716             for(p = 1; p < max_generations; p++){
717                 cout<<p<<" : "<<calc_avg_fitness()<<endl;
718                 avg_fitness[p] = calc_avg_fitness();
719                 next_Generation();
720                 Chromosome temp = find_fittest();
721                 if (best_chromosome.c_fitness > temp.c_fitness){
```

```
722                 best_chromosome = temp;
723             }
724             fittest_per_run[qq][p]=temp.c_fitness;
725         }
726         if (best_chromosome.c_matches == table_rows){
727             outputFile<<"=============== Run number : "<<qq+1<<" ====================="<<
                endl;
728             best_chromosome.printChromosome();
729             outputFile<<"----> "<<best_chromosome.c_fitness<<" "<<best_chromosome.
                c_generation<<"(Solution found!)"<<endl;
730             success_iter.push_back(best_chromosome.c_generation);
731             success_cnt++;
732             outputFile<<endl;
733         }
734         else {
735             outputFile<<"=============== Run number : "<<qq+1<<" ====================="<<
                endl;
736             best_chromosome.printChromosome();
737             outputFile<<"----> "<<best_chromosome.c_fitness<<" "<<best_chromosome.
                c_generation<<"(No solution found!)"<<endl;
738             outputFile<<endl;
739         }
740     }
741     outputFile<<endl;
742     int summ = 0;
743     for(int pp=0;pp<success_iter.size();pp++){
744         summ = summ+success_iter.at(pp);
745     }
746     outputFile<<"Average generation: "<<(double)summ/num_runs<<endl;
747     outputFile<<"Minimum : "<<min_max(success_iter,"min")<<endl;
748     outputFile<<"Maximum : "<<min_max(success_iter,"max")<<endl;
749     outputFile<<"Success rate (%) : "<<100*success_cnt/num_runs<<endl;
750     vector<double> avgs;
751     for(int j=0;j<max_generations;j++){
752         double sum_ = 0;
753         for(int i=0;i<num_runs;i++){
754             sum_=sum_+fittest_per_run[i][j];
755         }
756     avgs.push_back((double)sum_/num_runs);
757     }
758     for(int r = 0; r < max_generations; r++){
759         oFile<<avgs.at(r)<<"\n";
```

```
760         }
761     }
762 }
763 /*--------------------main function--------------------------------*/
764 int main(){
765     /*
766     The input dataset is structured as follows:
767     no. rows
768     no. cols
769     no. inputs
770     no. outputs
771     no. terminals, including 0 and 1
772     data*/
773     srand(time(0));
774     ifstream inputFile;
775     string filename_in;
776     cout << "Please enter the name of your input file: ";
777     cin >> filename_in;
778 
779     inputFile.open(filename_in.c_str());
780     if (inputFile.fail()) {
781             cout << "Input file could not be opened. Try again." << endl;
782             return 1;
783     }
784     cout << "Please enter the name of your output file: ";
785     cin >> filename_out;
786     outputFile.open(filename_out.c_str());
787     if (outputFile.fail()) {
788             cout << "Output file could not be opened. Try again." << endl;
789             return 1;
790     }
791     string avg_fit;
792     cout << "Please enter the name of your avg file: ";
793     cin >> avg_fit;
794 
795     oFile.open(avg_fit.c_str());
796         if (oFile.fail()) {
797             cout << "Avg file could not be opened. Try again." << endl;
798             return 1;
799         }
800     cout << "Choose between IGEP and GEP - igep/gep?: ";
801     cin >> reuse;
```

```
802     int rows = 0, inputs = 0, cols = 0, num_terms;
803     inputFile >> rows;
804     inputFile >> cols;
805     inputFile >> inputs;
806     inputFile >> outputs;
807     inputFile >> num_terms;
808     table_rows = rows;
809     table_cols = cols;
810     num_inputs = inputs;
811     num_terminals = num_terms/*-2*/;/*exclude 0 and 1 from the terminal set*/
812     num_functions = functions.size();//number of functions
813
814     if(inputFile.eof()){
815         cout << "Error reading input file contents." << endl;
816         return 1;
817     }
818     string *names = new string[table_rows + 1];
819     bool *values = new bool[table_rows];
820
821     for(int i = 0; i < table_rows + 1; i++){
822         inputFile >> names[i];
823     }
824     terms = names[0];
825     string terms_excl_1_0 = terms.substr(0,terms.size()-2);
826     fs_ter_gr = functions + terms/*terms_excl_1_0*/;
827
828     if (reuse == "igep"){
829         fs_ter_gr = fs_ter_gr + ref_genes.substr(0,ref_genes.size()-1);
830     }
831     fs_ter_gr_len = fs_ter_gr.size();
832     bool table[table_rows][table_cols];
833
834     for (int m = 1; m < table_rows + 1; m++) {
835         string binaries = names[m];
836         for (int n = 0; n < cols; n++) {
837                 if (binaries.at(n) == '0') {
838                     table[m - 1][n] = false;
839                 }
840                 else{
841                     table[m - 1][n] = true;
842                 }
843         }
```

```
844     }
845     truth_table_pointer = &table[0][0];
846     first_output_col = num_inputs;
847     int arity = 2;
848     tail_length = head_length * (arity - 1) + 1;
849     gene_length = head_length + tail_length;
850     outputFile<< "Head size: "<<head_length; outputFile<<"\n";
851     outputFile<< "Arity : "<<arity; outputFile<<"\n";
852     outputFile<< "Tail size: "<<tail_length; outputFile<<"\n";
853     outputFile<< "Gene size: "<<gene_length; outputFile<<"\n";
854     outputFile<< "Terminals & functions: "<<fs_ter_gr_len;outputFile<<"\n";
855     outputFile<< "Number of genes: "<<num_genes; outputFile<<"\n";
856     outputFile<<"# Table rows : "<<table_rows; outputFile<<"\n";
857     outputFile<<"# Table cols : "<<table_cols; outputFile<<"\n";
858     outputFile<<" Logic function : "<<filename_in; outputFile <<"\n";
859     outputFile<<"
          ------------------------------------------------------------------------";
860     outputFile<<"\n";
861     outputFile<<endl;
862     createPopulation();
863     outputFile.close();
864     oFile.close();
865 }
```

Bibliography

[1] N.A. AL-Saati and N. AL-Assady. Improving gene expression programming method. Available at `http://www.computerscience.uomosul.edu.iq.files/pages/page_5985249.pdf` (2014/01/28).

[2] R. Aler, D. Camacho, and A. Moscardini. The effects of transfer of global improvements in genetic programming. *Computing and Infomatics*, 23(4):377–394, 2004.

[3] S.M. Ashik-Eftakhar, S.K. Mahbub-Habib, and M.M.A. Hashem. Evolutionary design of digital circuits using genetic programming. In *Proceedings of the 3rd International Conference on Electrical, Electronics and Computer Engineering (ICEECE), Dhaka, Bangladesh, 22-24 December 2003*, pages 231–236. Cornel University Library, 2003.

[4] J. Clegg, J.A. Walker, and J.F. Miller. A new crossover technique for cartesian genetic programming. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 1580–1587. ACM Press, 2007.

[5] C.A. Coello-Coello, A.D. Christiansen, and A. Hernández-Aguirre. Towards automated evolutionary design of combinational circuits. *Computers and Electrical Engineering*, 27(1):1–28, 2001.

[6] W. Deng, P. He, and Z. Huang. Multi-expression based gene expression programming. In *Proceedings of 2013 Chinese Intelligent Automation Conference*, pages 439–448, 2013.

[7] E. Dufourq and N. Pillay. A preliminary study on the reuse of subtrees within decision trees in a genetic programming context for data classification. In *Third World Congress on Information and Communication Technologies (WICT)*, pages 287–292, 2013.

[8] C. Ferreira. Gene expression programming: A new adaptive algorithm for solving problems. *Complex Systems*, 13(2):87–129, 2001.

[9] C. Ferreira. Genetic representation and genetic neutrality in gene expression programming. *Advances in Complex Systems*, 5(4):389–408, 2002.

[10] Q. Freeman. Evolutionary art with multiple expression programming. Available at `http://www.math.uaa.alaska.edu/~afkjm/studentprojects/quentin_freeman_files/` (2015/03/15).

[11] F.Z. Hadjam, C. Moraga, and M.K. Rahmouni. Evolutionary design of digital circuits using improved multi-expression programming (IMEP). *Mathware and Software Computing*, 14:103–123, 2007.

[12] A. Hernández-Aguirre, B.P. Buckles, and C.A. Coello-Coello. Gate-level synthesis of boolean functions using binary multipliers and genetic programming. *IEEE*, pages 675–682, 2000.

[13] S. Karakatic, V. Podgorelec, and M. Hericko. Optimization of combinational logic circuits with genetic programming. *Electronics and Electrical Engineering*, 19(7):86–89, 2013.

[14] M.A. Karim and X. Chen. *Digital design: Basic concepts and principles.* CRC Press, Boca Raton, Florida, 2007.

[15] J.R. Koza. *Genetic programming: On the programming of computers by means of natural selection.* MIT Press, Cambridge, Massachusetts, 1st edition, 1992.

[16] J.R. Koza, D. Andre, F.H. Bennett III, and M.A. Keane. Use of automatically defined functions and architecture-altering operations in automated circuit synthesis with genetic programming. In *Proceedings of the 1st annual conference on genetic programming*, volume 13, pages 132–149. MIT Press Cambridge, MA, USA, 1996.

[17] T.R. Kuphaldt. *Lessons in Electric Circuits, Volume IV - Digital.* Tony R Kuphaldt, 4th edition, 2007.

[18] E.G. López, R. Poli, and C.A. Coello-Coello. Reusing code in genetic programming. In *7th European Conference , EuroGP 2004, Coimbra, Portugal, April 2004 proceedings*, 2004.

[19] J.F. Miller, editor. *Cartesian Genetic Programming*, pages 17–43. Springer-Verlag, Berlin, 2011.

[20] M. Oltean. Evolving reversible circuits for the even-parity problem. Available at `http://www.cs.ubbcluj.ro/~moltean/oltean_rev.pdf` (2015/08/15), 2005.

[21] M. Oltean. Multi expression programming, Technical Report. Available at `http://www.mep.cs.ubbcluj.ro/papers.html` (2015/04/22), 2006.

[22] M. Oltean and C. Grosan. A comparison of several linear genetic programming techniques. *Advances in Complex Systems*, 4(14):285–313, 2003.

[23] M. Oltean and C. Grosan. Evolving digital circuits using multi expression programming. In *Proceedings of the 2004 NASA/DoD Conference on Evolvable Hardware, Seattle, Washington (USA), 24-26 June 2004*, pages 87–90. IEEE Press, NJ, 2004.

[24] M.A. Qureshi. *The Evolution of Agents.* PhD thesis, Department of Computer Science University College London, Department of Computer Science University College London, 2001. https://www.cl.cam.ac.uk/ jac22/otalks/aqureshi-draft.pdf (2015/04/26).

[25] C. Reis and J.A. Tenreiro Machado. An evolutionary approach to the synthesis of combinational circuits. In *Proceedings of IEEE International Conference on Computational Cybernetics, Siófok, Hungary, August 29 - 31, 2003.*

[26] L. Rutkowski. *Computational Intelligence: Methods and Techniques.* Springer-Verlag, Berlin, 2008.

[27] J.E. Savage. *Models of Computation: Exploring the Power of Computing.* Addison-Wesley Longman, Incorporated, reprint edition, 1998.

[28] L. Sekanina and Z. Vašíček. Evolutionary computing in approximate circuit design and optimization. In *1st Workshop on Approximate Computing (WAPCO 2015)*, pages 1–6, 2015.

[29] S.N. Sivanandam and S.N. Deepa. *Introduction to Genetic Algorithms.* Springer, 1st edition, 2007.

[30] W.-H. Steeb. *The nonlinear workbook*. World Scientific Publishing Co. Pte. Ltd, London, UK, 5th edition, 2011.

[31] Y. Suttasupa, S. Rungraungsilp, S. Pinyopan, P. Wungchusunti, and P. Chongstitvatana. A comparative study of linear encoding in genetic programming. In *2011 Ninth International Conference on ICT and Knowledge*, 2011.

[32] C. Wang, J. Zhang, S. Wu, F. Zhang, and J.G. Tromp. An improved gene expression programming based on niche technology of outbreeding fusion. *Infomatica*, 41(1):25–30, 2017.

[33] X. Yan, W. Wei, Q. Liang, C. Hu, and Y. Yao. Designing electronic circuits by means of gene expression programming II. *Evolvable Systems: From Biology to Hardware*, 4684:319–330, 2007.

[34] B. Yuan and M. Gallagher. An improved small-sample statistical test for comparing the success rates of evolutionary algorithms. In *Proceedings of the 11th annual conference on genetic and evolutionary computation*, pages 172–175. ACM New York, USA, 1997.